Volume 15 Number 4

ACTA CYBERNETICA

Szeged, 2002

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Contributions are accepted for review with the understanding that the same work has not been published elsewhere.

Manuscripts must be in English and should be sent in triplicate to any of the Editors. On the first page, the *title* of the paper, the *name(s)* and *affiliation(s)*, together with the *mailing* and *electronic address(es)* of the author(s) must appear. An *abstract* summarizing the results of the paper is also required. References should be listed in alphabetical order at the end of the paper in the form which can be seen in any article already published in the journal. Manuscripts are expected to be made with a great care. If typewritten, they should be typed double-spaced on one side of each sheet. Authors are encouraged to use any available dialect of TEX.

After acceptance, the authors will be asked to send the manuscript's source TEX file, if any, on a diskette to the Managing Editor. Having the TEX file of the paper can speed up the process of the publication considerably. Authors of accepted contributions may be asked to send the original drawings or computer outputs of figures appearing in the paper. In order to make a photographic reproduction possible, drawings of such figures should be on separate sheets, in India ink, and carefully lettered.

There are no page charges. Fifty reprints are supplied for each article published.

Publication information. Acta Cybernetica (ISSN 0324–721X) is published by the Department of Informatics of the University of Szeged, Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. For 2002 Numbers 3–4 of Volume 15 are scheduled. Subscription prices are available upon request from the publisher. Issues are sent normally by surface mail except to overseas countries where air delivery is ensured. Claims for missing issues are accepted within six months of our publication date. Please address all requests for subscription information to: Department of Informatics, University of Szeged, H-6701 Szeged, P.O.Box 652, Hungary. Tel.: (36)-(62)-420-184, Fax:(36)-(62)-420-292.

URL access. All these information and the contents of the last some issues are available in the Acta Cybernetica home page at http://www.inf.u-szeged.hu/local/acta.

L. Budach
University of Postdam
Department of Computer Science
Am Neuen Palais 10
14415 Postdam, Germany

H. Bunke
Universität Bern
Institut für Informatik und
angewandte Mathematik
Längass strasse 51.
CH-3012 Bern, Switzerland

B. Courcelle
Universitè Bordeaux-1
LaBRI, 351 Cours de la Libèration
33405 TALENCE Cedex
France

J. Demetrovics
MTA SZTAKI
Budapest, Lágymányosi u. 11.
H-1111 Hungary

B. Dömölki
IQSOFT
Budapest, Teleki Blanka u. 15-17.
H-1142 Hungary

J. Engelfriet
Leiden University
LIACS
P.O. Box 9512, 2300 RA Leiden
The Netherlands

Z. Ésik
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

L. Lovász
Eötvös Loránd University
Department of Computer Science
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

G. Păun
Institute of Mathematics
Romanian Academy
P.O.Box 1-764, RO-70700
Bucuresti, Romania

A. Prékopa
Eötvös Loránd University
Department of Operations Research
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

A. Salomaa
University of Turku
Department of Mathematics
SF-20500 Turku 50, Finland

L. Varga
Eötvös Loránd University
Department of General Computer Science
Budapest, Pázmány Péter sétány 1/c.
H-1117 Hungary

H. Vogler
Dresden University of Technology
Department of Computer Science
Foundations of Programming
D-01062 Dresden, Germany

G. Wöginger
Department of Matematics
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

Preface

This issue of *Acta Cybernetica* contains a selection of papers whose preliminary versions appeared in the proceedings of the Seventh Finno-Ugoric Symposium on Programming Languages and Software Tools (SPLST). The symposium was held in Szeged, Hungary, on June 15–16, 2001. The main aim of this symposium series is to provide a forum in every second year for computer scientists living in Estonia, Finland, and Hungary to exchange ideas and present their current works and also to offer young scientists the possibility of demonstrating their results at an international scientific meeting.

After the symposium the authors were invited to submit completed versions of their papers for this special issue. All submitted papers were then subjected to the normal refereeing process of *Acta Cybernetica*. Altogether 20 manuscripts were submitted, out of which 14 have been accepted.

We thank the authors, the members of the Program Committee, and the referees for their help in the preparation of this issue.

Tibor Gyimóthy
Chairman of SPLST 01

Zoltán Fülöp
Managing Editor of AC

Acta Cybernetica 15 (2002) 481–487.

Human Cognition of Complex Thought Patterns

Harry M. Sneed*

Abstract

How much is our perception of the present determined by our experience of the past?

1 Theories on Human Cognition

The Scottish philosopher David Hume was one of the first to point out that our ability to distinguish between cause and effect is not all due to an objective perception of reality, but rather to build in notions of what we expect [1]. Hume, being a sceptic, wanted to take issue with the empiricist John Locke who believed humans could actually perceive objective reality and therefore determine causality [2]. Hume disputed this assertion, claiming instead that causality has no objective basis. It is not derived from our empirical perception. Instead it comes from reasoning and reasoning is subjective. In other words, Hume questioned our ability to see things as they are, an opinion later picked up and expounded upon by the modern philosopher John Kemeny in respect to modern science [3].

Immanuel Kant, the great German philosopher assumed Humes position and brought it to a logical conclusion in his famous essay on "Die Kritik der reinen Vernunft". According to Kant the so called natural order of the universe only exist in our minds. The human mind is the source of all order. Kant states "Since it is our thought which forms the order of reality, one can not say that our perception is determined by the objects we perceive, rather one must say that the objects we perceive are determined by our notion of how they should be " [4] This criticism of our objective reasoning power was in direct contradiction to the prevailing philosophy of the times as propagated by the French philosophers based on the teachings of Descartes and others [5].

The basic thesis of Kant is that there is no such thing as an objective perception of reality. All perception is formed to fit patterns of thought which exist a priori in our minds. Thus, all perception is by nature subjective. Assuming this to be true, the next question is where do these thought patterns come from, how do they originate?

*Institut für Wirtschaftsinformatik, University of Regensburg, Germany,
e-mail: Harry.Sneed@T-Online.de

Kant answered this question on a religious ethical basis, claiming that mankind is endowed with God given thought patterns which allow him to perceive reality as it should be. Kant believed in natural law and in a universal reasoning power which allows us to distinguish between right and wrong. Today, biologists would claim that this reasoning power is inherited and behaviouralists would claim that it is formed by our environment. Most likely it is influenced by both the experiences of our ancestors as well as by our own early experiences, as pointed out by Bertrand Russel in his famous essay on the limits of human knowledge [6].

A good example of priori thought patterns is natural language. In describing reality man must assign what he perceives to words he knows. If there is no word, then the object may exist, but it can not be expressed. We only distinguish objects if we are able to assign them to a pre existing notion and pre existing notions are provided by our language. Therefore, our language determines how we perceive things. In case of simple objects, all languages are similar. There is usually a word which one attaches to the object. However, in case of complex objects, languages can differ extremely in how they assign meaning to these objects. They use different thought schemes, which makes it so difficult to translate such concepts. [7] It is here where environment comes into play. The development of natural languages is influenced by environment in which they evolve. It is evident that the language of Eskimos will differ from the language of natives of a tropical island. It is also evident that the language of an agricultural community where people are working in the fields will differ from the language of an industrial community where people are working in factories. Children inherit word and notions, i.e. thought patterns from their parents. As they grow up and have their own experiences these inherited thought patterns are enhanced or even overridden by new context dependent thought patterns formed by another environment. However, one is never free of the original thought patterns one inherits. They are part of our apriori cognitive process and influence everything we perceive. This is really what Kant refers to as our God given power of reasoning [8].

A modern advocate of Kant's theory of cognition is Paul Feyerabend. However, Feyerabend does not believe in any God given power of reasoning not even in a natural law. Instead, Feyerabend claims that natural laws are the result of human conventions which have evolved over time, i.e. something like Locke's social contract. Feyerabend vehemently objects to anything like an objective perception of reality. Nothing can really be proven, since all proofs are based on conventions and all conventions are man made. Thus, all cognition is relative to the prevailing mode of perception. There is no such thing as a single universal view of complex subject matter. Even the language of mathematics is a matter of convention. There are no self-evident truths, as demonstrated by the assertion disproved by the German mathematician Goldberg. Thus, not even mathematical propositions are self-evident, let alone computing algorithms or programming language constructs. The Feyerabend school is referred to as "Constructivism" [9].

The fact that cognition is a result of predetermined thought patterns does not necessarily imply that there is no influence of our impressions upon our cognitive

abilities. The theory of evolution also applies to human perception. Our minds are conditioned by the environment in which we live. The ability of animals to perceive their environment is the result of a long evolutionary process and it differs from species to species. Therefore, bats and cats have totally different mental representations of their environment just as do men and dogs. Dogs have evolved to react to what they hear and smell. Men have evolved to react to what they think they see. In this respect our minds have been conditioned to see things as we would believe them to be. Plato compares human cognition with the perception of shadows on a wall. It is not reality we perceive, but only images of this reality which we have been conditioned to recognise [10].

2 Programmer Cognition of complex Programs

Now what might the theory of human cognition and the school of "constructivism" have to do with program comprehension? The answer to this question should be obvious. Programs are representations of complex thought patterns. These patterns are determined to a great extend by the languages or methods in which they are expressed. In fact it would be impossible for them to be expressed without a language. On the other hand, they can only be comprehended by minds conditioned to discern thought patterns in that language. Just as different animal species have developed different means of perceiving their environment, different programmer species have been conditioned to comprehend programs in different ways [11]. A COBOL Programmer will perceive a problem in a different mode than will a C Programmer. Ergo, there is no such thing as a universal approach to program comprehension as long as the minds of programmers are being conditioned by different languages and different methods. The thought patterns of programmers come from their experience in programming, especially from their early experiences. This is where the universities have a tremendous responsibility in implanting cognition patterns similar to the responsibility of mothers in instilling behavioural patterns.

A good example of this is the transition from procedurally oriented to object-oriented programs and designs. A programmer who has been conditioned to think in procedural terms will not been able to comprehend an object oriented program or program design no matter how well it is documented. The programmer will not be able to relate the constructs he perceives with the apriori constructs in his mind. The same of course will hold in the inverse direction. Minds conditioned to comprehend object-oriented constructs will have difficulty in comprehending procedural ones. This is why it is so difficult to get young college trained programmers to maintain old programs. They simply cannot relate to the concepts contained therein. These problems have been documented by Fayad and others [12].

Unfortunately it is still believed by many, especially by persons in management, that if programs are documented well enough, they should be comprehensible to anyone including themselves. This is one of the many myths introduced to the world by great simplifiers like James Martin who also advocated that programming can be done without programmers [13].

A complex Assembler program can only be comprehended by a person familiar with the semantics of Assembler. Even if the program is converted to a higher level language, it will still retain its Assembler semantics. It simply becomes an Assembler program with a COBOL or C syntax and it will take someone conditioned to think as an Assembler programmer to fully comprehend it. No amount of diagramming techniques will ever suffice to make the program comprehensible to anyone without pre existing Assembler thought patterns [14].

The same applies to higher level languages. A program written in a 4GL language such as NATURAL reflects the thought patterns determined by that language with constructs such as map driven and database driven loops, constructs which do not exist in standard 3GL languages. Thus, they can only be comprehended by minds conditioned to think in these terms, i.e. minds which have been exposed to 4GL languages.

The deluge of diverse programming languages and design methods has led to a highly fragmented programming community, where hardly anyone is able to comprehend the work of others. The tower of Babel is being reconstructed in the Software community. Documentation and annotation is of no help unless of course the subject is familiar with the solution domain. Even if one is familiar with the problem, he will still have difficulties comprehending the solution if he is not familiar with the language and method used, e.g. one conditioned to think procedurally will not be able to comprehend an object-oriented solution. Ergo, as long as languages and solution approaches are continually evolving, the program comprehension problem will increase regardless of the quality of documentation.

The contention here is that documentation, whether it be static or dynamic, tabular or graphic, is part of the solution domain. Therefore, it cannot really promote comprehension unless the subject has preconditioned thought patterns which can order and interpret the information contained in the documentation. Human minds must be conditioned to think in terms of the methods used to solve problems in order to understand the problem solutions. Thus program comprehension is really a matter of thought patterns installed in the minds of programmers and analysts by training and experience. These thought patterns are shaped by the languages and methods to which they have been exposed.

Von Mayrhauser states that "program comprehension uses existing knowledge to acquire new knowledge that ultimately meets the goals of a code cognition task. This process references both existing and newly acquired knowledge to build a mental model of the software under consideration" [15]. The contention here is that the existing knowledge, i.e. the apriori mental concepts, must be well above 50 % of the total knowledge required to understand complex systems. The maintainer must already have expectations as to how a system should be constructed and how it should behave in a given context. That implies that he has been exposed to both the solution and the problem domain.

The emphasis here is on exposure. Anyone who has ever dealt with programming knows that it is never learned passively by just talking or reading about it. Thought patterns can only be installed by actively applying them. This applies to natural

languages as well as to programming languages. Thus, complex programs can only be comprehended by persons who have experienced the method and language with which the programs have been implemented. Documentation may help to locate and understand the role of program artifacts, but only if they are in the domain to which the subject has been conditioned to comprehend [16].

3 Conclusions to be drawn

What conclusions can be derived from these observations. The first is that we in the program comprehension community should not over estimate the role of documentation. It can be helpful, but it is no substitute for preconditioned knowledge. An experienced C++ programmer will always understand a complex C++ program better than an experienced COBOL programmer no matter how much documentation the latter has. Documentation can not make up for the lack of apriori thought patterns.

Another conclusion is that one will not understand what a program is doing unless one is familiar with the subject area. He may understand the solution but he will never understand the problem. Also here documentation is of limited help, since documentation assumes that the reader is familiar with the terminology and frames of reference used to describe that subject area. Thus someone unfamiliar with aerodynamics but experienced in C++ may understand how a C++ program for calculating the affects of wind currents is functioning but will never understand why. This lack of knowledge will be detrimental if he is called upon to change the algorithm.

The final conclusion is that there is no substitute for apriori knowledge when it comes to comprehending programs. The maintenance programmer should be familiar with both the problem area and the solution domain, in order to be effective. This required degree of familiarity can only be obtained by conditioning the mind through practical experience, i.e. to exposure to the subject area.

4 Actions to be taken

This being the case, there are several ways in which industry can improve the software maintenance situation. The first is through standardisation. There should be less languages and less methods and these should be accessible to a wider range of programmers. Proprietary languages should be strictly avoided if at all possible. The productivity gain made by using Oracle Forms or Power Builder in development is later negated by the problems of maintaining or reusing such solutions. UML - Unified Modelling Language - from the OMG is certainly a step in the right direction just as is CORBA-IDL and ANSI C++.

The second way is by retaining experienced personnel in software maintenance. Persons with knowledge of both the subject area and the solution space are invaluable assets to the organisation. They should be cultivated and valued. If they must

move on to other work, there should be a long transition period for them to pass their work knowledge on to their successors.

A third way is by getting greater user involvement in the software maintenance process. If the programmers are not so familiar with the application domain, it is essential that they work together with an end user who has this knowledge and can pass it on. This also means that the user representative is able to help in locating errors and in assessing the impact of changes.

A fourth way is through continuity. By continually changing their languages and design methods user organisations only disrupt the learning process. An organisation should be extremely careful in the selection of a design method and programming language. Once it has committed itself, it should then stick to that language and design approach as long as possible so as to profit from existing thought patterns. Software Managers must learn like Odysseus to close their ears to the wails of the modern day sirens – the Software Case vendors – who would have them believe that all of their problems could be solved by introducing a new development technology.

A fifth and final way is through training. New maintenance programmers should be trained both in the subject area and in the languages and methods used by the organisation. This training should go beyond formal presentation. The subjects must have the opportunity to condition their minds through participation in exercises designed to install thought patterns. This may be expensive and time consuming, but it will be well worth it in improving maintenance productivity. Older programmers should even be given a year of absence to condition their minds to a new technology such as distributed objects.

In summary, the point presented in this paper is that program comprehension is determined more by mind conditioning than by any means of documentation or annotation. Program documentation and annotation is helpful, but only if the thought patterns represented in the documentation are familiar to the subjects. Knowledge may be extracted from software artifacts, but is will only be accessible to persons conditioned to recognise that kind of knowledge. Thus, the thought patterns used in the programs must exist apriori in the minds of the programmers in order for them to associate the two, i.e. what they think they see with what they think they know. This type of pattern watching is the essence of human cognition and program comprehension is after all just another cognition problem.

References

[1] Hume,D. : "An Enquiry Concerning Human Understanding" Open Court Publishing, La Salle, J II., 1946

[2] Locke, J. : "An Essay Concerning Human Understanding" Aldine Press Letchworth G.B., 1961

[3] Kemeny, J. : A Philosopher Looks at Science", van Nostrand, Princeton, 1959

[4] Kant, I. : "Die Kritik der reinen Vernunft" in Geschichte der Philosophie, Ed. Hans String, Bertelsmann Verlag, Stuttgart,1962

[5] Descartes, R. : "Discours de la methode" in History of Materialism, Ed. F. Lange, Simon and Schuster, New York, 1961

[6] Russel, B. : "Human Knowledge - its Scope and Limits", Simon and Schuster, New York, 1948

[7] Stevenson, C.L. : "Ethics and Human Language" Yale University Press, New Haren, 1944

[8] Kant, I. : "Metaphysik der Sitten" in Werke von I. Kant, Ed.E. Cassirer, Herter Verlag, Berlin, 1912

[9] Feyerabend, P. : "Against Method-outline of an anarchistic theory of Knowledge" Atlantic Highlands, Princeton, 1974

[10] Platon, G. : "Der Staat", Artemis Verlag, Zrich, 1960

[11] Dijkstra, E. : "On the Cruelty of really Teaching Computer Science" in Comm. of ACM. Vol. 32, No.12 Dec. 1989

[12] Fayad, M./Tsai, W./Fulghim, L. : "Transition to Object-oriented Software Development", Comm. of ACM. Vol.39, No. 2, Feb. 1996

[13] Martin, J. : "Application Development without Programmers", Prentice-Hall, Englewood cliffs, 1981

[14] Martin, J. : "Diagramming Standards for Analysts and Programmers", Prentice-Hall, Englewood cliffs, 1987

[15] von Mayrhausen, A./Vans, M. : "Program Comprehension during Software Maintenance and Evolution" in IEEE Computer, Aug. 1995

[16] van der Meulen, M./Hage,J. :"Human Factors in Software Maintenance" Report of ESPRIT Project MACS, Maastricht, 1992

Acta Cybernetica **15** (2002) 489–508.

Handling Pointers and Unstructured Statements in the Forward Computed Dynamic Slice Algorithm

Csaba Faragó* and Tamás Gergely†

Abstract

Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be classified as a static slicing or dynamic slicing type. In applications such as debugging the computation of dynamic slices is more preferable since it can produce more precise results. In a recent paper [5] a new so-called "forward computed dynamic slice" algorithm was introduced. It has the great advantage compared to other dynamic slice algorithms that the memory requirements of this algorithm are proportional to the number of different memory locations used by the program, which in most cases is much smaller than the size of the execution history. The execution time of the algorithm is linear in the size of the execution history. In this paper we introduce the handling of pointers and the jump statements (`goto`, `break`, `continue`) in the `C` language.

1 Introduction

Program slicing methods are widely used for debugging, testing, reverse engineering and maintenance (e.g. [3], [7], [2], [4]). A slice consists of all statements and predicates that might affect the variables in a set V at a program point p [8]. A slice may be an executable program or a subset of the program code. In the first case the behaviour of the reduced program with respect to a variable v and program point p is the same as the original program. In the second case a slice contains a set of statements that might influence the value of a variable at point p. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*) or compute those statements which influence the value of a variable occurrence for a specific program input (*dynamic slice*).

In many applications (e.g. debugging) the computation of dynamic slices is more preferable since it can produce more precise results (i.e. the dynamic slice is smaller than the static one). In this paper we will focus on dynamic slicing.

*Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanúk tere 1., H-6720 Szeged, Hungary, e-mail: csaba@petra.hos.u-szeged.hu

†Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanúk tere 1., H-6720 Szeged, Hungary, +36 62 544143, e-mail: gertom@inf.u-szeged.hu

In [5] Tibor Gyimóthy, Gábor Forgács and Árpád Beszédes introduced a method for the forward computation of dynamic slices (i.e. at each iteration of the process, slices are available for all variables at the given execution point). However, the method presented was applicable only to very simple programs (with one procedure, scalar variables and simple assignment statements only). In [9] the handling of the procedures and the implementation of the algorithm were shown. In this paper we show how to handle pointers and the jump statements in the C programs. In addition to the `goto` statement it solves the problem of `break` and `continue` statements, which can be regarded as special cases of the `goto` statement. The handling of the `switch-case-dafault` statement is also mentioned.

The paper is organized as follows. After discussing the background of slicing, the "forward computed dynamic slice" method is introduced. The handling of pointers and jump statements are then elaborated on in Sections 3 and 4. Finally, we give a summary of what we have done so far.

2 Forward computing of the dynamic slice

2.1 Original algorithm

In some applications static program slices contain redundant instructions. This is the case for debugging, for instance, where we have dynamic information as well. Hence debugging may require smaller slices, which improves the efficiency of the bug finding process ([1], [6]). The goal of the introduction of dynamic slices was to determine more precisely those statements that may contain program bugs, assuming that the failure has occurred for a given input.

Consider the example program in Figure 1. The static slice of this code with respect to the variable s at vertex 12 contains all the statements.

Prior to the description of a new dynamic slice algorithm we introduce some basic concepts and notations.

A feasible path that has actually been executed will be referred to as an *execution history* and denoted by EH. Let the input be $a = 0$, $n = 2$ in the case of our example. The corresponding execution history is $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7,$ $8, 10, 11, 7, 12\rangle$. We can see that the execution history contains instructions which come in the same order as they have been executed, so $EH(j)$ gives the serial number of the instruction executed at the j^{th} step, referred to as *execution position* j.

To distinguish between multiple occurrences of the same instruction in the execution history we make use of the notion of *action*. It is a pair (i, j) which is written as i^j, where i is the serial number of the instruction at the execution position j. For example 12^{15} is the action for the output statement of our example for the same input as above.

The *dynamic slicing criterion* is a triplet $(\mathbf{x}, i^j, V)$ where $\mathbf{x}$ denotes the input, i^j is an action in the execution history, and V is a set of the variables. For a slicing criterion a dynamic slice can be defined as the set of statements which may affect

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
1.     scanf("%d", &n);
2.     scanf("%d", &a);
3.     i = 1;
4.     s = 1;
5.     if (a > 0)
6.       s = 0;
7.     while (i <= n) {
8.       if (a > 0)
9.         s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

Figure 1: Example program

the values of the variables in V.

We apply a program representation which only considers the definition of a variable, and use of variables, and direct control dependences. We refer to this program representation as a *D/U program representation.* An instruction of the original program has a D/U expression of the form:

$$i.\, d:\, U,$$

where i is the serial number of the instruction, and d is the variable that gets a new value from the instruction in the case of assignment statements. For an output statement or a predicate d denotes a newly generated "output variable"- or "predicate-variable"-name of this output or predicate, respectively (see the example below). Let $U = \{u_1, u_2, ..., u_n\}$ such that any $u_k \in U$ is either a variable that is used at i or a predicate-variable from which instruction i is (directly) control dependent. Note that there is at most one predicate-variable in each U. (If the *entry* statement is defined, there is exactly one predicate-variable in each U.)

Our example has a D/U representation shown in Figure 2.

Here $p5$, $p7$ and $p8$ are used to denote predicate-variables and $o12$ denotes the output-variable, whose value depends on the variable(s) used in the output statement.

Now we are ready to derive the dynamic slice with respect to an input and

$i.$	d	:	U
1.	n	:	$\emptyset$
2.	a	:	$\emptyset$
3.	i	:	$\emptyset$
4.	s	:	$\emptyset$
5.	$p5$	:	$\{a\}$
6.	s	:	$\{p5\}$
7.	$p7$	:	$\{i, n\}$
8.	$p8$	:	$\{p7, a\}$
9.	s	:	$\{s, p8\}$
10.	s	:	$\{s, p8\}$
11.	i	:	$\{i, p7\}$
12.	$o12$	:	$\{s\}$

Figure 2: D/U representation of the program

the related execution history based on the D/U representation of the program as follows. First, we process each instruction in the execution history starting from the first executed statement. Then after processing an instruction $i.\ d : U$, we derive a set $DynSlice(d)$ that contains all those statements which affect d when instruction i has been executed. By applying the D/U program representation the effect of data and control dependences may be treated in the *same way*. After an instruction has been executed and the related *DynSlice* set has been derived, we determine the *last definition* (serial number of the instruction) for the newly assigned variable d denoted by $LS(d)$. Put simply, the last definition of variable d is the serial number of the instruction where d is last defined (considering the instruction $i.\ d : U$, $LS(d) = i$). Clearly, after processing the instruction $i.\ d : U$ at the execution position j each $LS(d)$ has the value i for each subsequent executions until d is redefined next time. We also use $LS(p)$ for predicates, which denotes the last definition (evaluation) of predicate p. For example, if $EH(10) = 7$ (the current action is 7^{10}) then $LS(d) = 7$.

Now the dynamic slices can be determined as follows. Assume that we are running a program having an input t. After an instruction $i.\ d : U$ is executed at position p, $DynSlice(d)$ contains just those statements involved in the dynamic slice for the slicing criterion $C = (t, i^p, U)$. *DynSlice* sets are determined by using the relation below:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big(DynSlice(u_k) \cup \{LS(u_k)\} \Big)$$

After $DynSlice(d)$ has been evaluated we determine $LS(d)$ for assignment and predicate instructions, i.e.

$$LS(d) = i$$

Note that this computation order is strict since when we determine $DynSlice(d)$, we have to consider whether $LS(d)$ occurred at a former execution position instead of p (like the program line `x = x + y` in a loop).

program DynamicSlice
begin
  Initialize LS and $DynSlice$ sets
  ConstructD/U
  ConstructEH
  for $j = 1$ **to** *number of elements in EH*
    the current D/U element is i^j. $d: U$
    $DynSlice(d) = \bigcup_{u_k \in U}(DynSlice(u_k) \cup \{LS(u_k)\})$
    $LS(d) = i$
  endfor
  Output LS and $DynSlice$ sets for the last definition of all variables
end

Figure 3: Dynamic slice algorithm

A formal version of the forward dynamic slice algorithm is presented in Figure 3. Note that the construction of the execution history is achieved by instrumenting the input program and executing this instrumented code. The instrumentation procedure is discussed in [9].

We will illustrate how the above method works by applying it to our example program in Figure 1 with the execution history ⟨ 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 ⟩.

During the execution the following values are returned:

Action	d	U	$DynSlice(d)$	$LS(d)$
1^1	n	$\emptyset$	$\emptyset$	1
2^2	a	$\emptyset$	$\emptyset$	2
3^3	i	$\emptyset$	$\emptyset$	3
4^4	s	$\emptyset$	$\emptyset$	4
5^5	$p5$	$\{a\}$	{2}	5
7^6	$p7$	$\{i,n\}$	{1,3}	7
8^7	$p8$	$\{p7,a\}$	{1,2,3,7}	8
10^8	s	$\{s,p8\}$	{1,2,3,4,7,8}	10
11^9	i	$\{i,p7\}$	{1,3,7}	11
7^{10}	$p7$	$\{i,n\}$	{1,3,7,11}	7
8^{11}	$p8$	$\{p7,a\}$	{1,2,3,7,11}	8
10^{12}	s	$\{s,p8\}$	{1,2,3,4,7,8,10,11}	10
11^{13}	i	$\{i,p7\}$	{1,3,7,11}	11
7^{14}	$p7$	$\{i,n\}$	{1,3,7,11}	7
12^{15}	$o12$	$\{s\}$	{1,2,3,4,7,8,10,11}	12

The final slice is the union of $DynSlice(o12)$ and $\{LS(o12)\}$. (See Figure 4)

```
#include <stdio.h>
int n, a, i, s;
void main()
{
 1.   scanf("%d", &n);
 2.   scanf("%d", &a);
 3.   i = 1;
 4.   s = 1;
 5.   if (a > 0)
 6.     s = 0;
 7.   while (i <= n) {
 8.     if (a > 0)
 9.       s += 2;
        else
10.       s *= 2;
11.     i++;
      }
12.   printf("%d", s);
}
```

Figure 4: The framed statements give the dynamic slice

2.2 Analysis of the algorithm

Let's analyze the duration and the memory requirement of the algorithm! It's very hard to figure out the exact requirements. We'll try to make an average-case analysis with referring to the worst-case, too.

First let's consider the duration. The initializations are approximately linear to the different memory locations. The DU construction is linear to the length of the executable source code. One can ask why don't we say that it is linear to the statements? The reason is that the duration of one step is dependendent to the length of the statement. For example it takes less time to build up the DU for `a=b+c` than for `a=b*c-f(b,b+c)-c`. The construction of the EH is linear to the execution of the original program. Unfortunately the constant multiplier hidden by "theta-notation" (it is used in analysis of the algorithms) is hardly predictable: it is dependent to the number of pointers etc., which can vary from zero up to the whole program. The duration of the first and the third pseudo-statement within the main `for` cycle is constant. The union statement's duration is critical within the algorithm, but unfortunately it is very hardly predictable. In worst case the U set can hold all the variables (i.e. different memory locations + pseudo-variables

(e.g. labels etc.)), and all the dynamic slices holds all the statements within the program. It this case the main cycle's duration is proportional to <execution history> * <memory locations> * <number of statements>. But in the most normal programs the size of the U set is not so big, in most cases it holds about 4-5 elements. There exist not too much such statement where the U set contains more than 10 elements. A dynamic slice in most cases contains not too much statement, but it seems in many cases it is linear to the size of the program. The duration of the output depends to the numbers of slice criteria etc., but it is less than the countation, of course. According to these the average execution time of the algorithm is O(|EH|*|statements|+|memory|).

Now let's analyze the memory requirements. The most relevant memory requirement takes the storage of the temporal slice results, i.e. the U sets; the others (e.g. memory requirements of the initialization part etc.) can be ignored. In the worst case every variable (i.e. every memory location) contains all the statements, so in this case the memory requirement is O(<number of different memory locations> * <number of statements>). In fact it is very unlikely to use such a big memory, it is rather linearly proportional to the different memory locations used during the program with a bigger constant. At bigger programs in the most cases the memory requirement of the dynamic counting algorithm is linear to the memory requirement of the original program (with a bigger constant).

2.3 Extending the basic algorithm

In order to handle the pointers, the variables are identified by their addresses and not by their names. This approach has several good advantages.One is that it solves the problem of the variables with the same name but different program scope.

The address of a variable can only be determined dynamically after its declaration, but the DU is derived from the static source code. Hence there are two DU structures: a static DU which contains variable names, and a dynamically resolved DU (dynamic DU) which contains addresses. (Note that the dynamic DU may change during the program execution due to a change in variable address, pointer value, etc. The neccessary parts of the dynamic DU are computed at each step using the static DU and the (extended) execution history.)

In a C program there may be several variables present with the same name but in different scopes. The address of a variable with a specific name may depend on the scope of the expression where the variable is used. So the algorithm must keep track of the scopes and maintain a stack structure for each function in order to store the addresses of the variables. Each time a new scope is begun, a new address table is created at the top of the stack, and when a new variable declaration occurs, the name and address are recorded in this new table. For the address of a variable the address tables are searched from the top to the bottom of the stack of the actual function. When a scope leaved, the top element of the stack is discarded. The first element at the bottom of each such stack is the same: the address table of global variables (these can be accessed by every function).

3 Handling pointers

In this subsection the handling of the pointers, arrays and structures are described. The methods we introduce are for the C language, but the general idea may be applied to other languages too. When code is shown, the "common" parts of the code (such as function headers, includes) are hidden, as well as some function calls from the instrumented code.

3.1 Pointers

The address of a variable does not change in its scope, so after it is determined it can be used any number of times. But the value of a pointer can change at any time and must be determined every time the pointer occurs. Consider the following program code:

```
    int x, *p;
1.  x=1;
2.  p=&x;
3.  *p=2;
4.  print(x);
```

It is readily seen that only lines 2 and 3 affect the value of x in line 4, while the first one doesn't. Statically it is almost impossible to detect these kind of dependencies. Most of the static algorithms either include the whole program, or exclude lines 2 and 3 and include only those like the first line, which will produce incorrect slices.

Statically, only the following dependencies can be determined:

line	def	:	USE
1	x	:	$\emptyset$
2	p	:	$\emptyset$
3	$PTR1$	:	$\{p\}$
4	OUT	:	$\{x\}$

where $PTR1$ indicates that a memory location is defined via a pointer. Of course, this memory location depends on the value of the pointer itself. Using dynamic information the variables can be converted into memory locations. This means that addresses 01 and 02 can be used instead of variables x and p, respectively. Dynamically, the value of $PTR1$ is also known. Extracting the information from the execution history, the result is the following (dynamically resolved) DU:

step	line	def	:	USE	$DynSlice(def)$
1	1	01	:	$\emptyset$	$\emptyset$
2	2	02	:	$\emptyset$	$\emptyset$
3	3	01	:	{02}	{2}
4	4	OUT	:	{01}	{2,3}

The technical procedure for the extraction of the addresses is called program instrumentation. This means modifying the program code in such way that the program retains its original behaviour, but also generates some extra information (eg. execution history, runtime addresses, block information). The instrumented code is similar to the following (actually, it is slightly more involved):

```
    int x, *p;
    remember("x", &x);
    remember("p", &p);
1.  x=1;
2.  p=&x;
3.  *dump("PTR1", p)=2;
4.  print(x);
```

The `remember` function writes the address of the variable into the (extended) execution history. This information is used during the execution of the slicing algorithm to create the dynamic DU. The `dump` function writes the value of its second parameter (in this case pointer value), and simply returns it. In the static DU case the third line contains the pointer variable $PTR1$, which can be resolved within the algorithm to an address using the previously dumped pointer value.

3.2 Arrays

In the C language the arrays and the pointers are practically the same, and the conversion from one to the other is quite simple. The i^{th} element of an array `t`, denoted by `t[i]`, can be expressed as a pointer `*(t+i)`. Then, when an element of an array is referenced, it is treated as a pointer in the DU and then its address is written out. Consider the following example:

```
    int t[5];
    int *p;
1.  t[0]=1;
2.  *(t+1)=2;
3.  t[2]=2*t[1];
4.  print(t[2]);
```

The static DU of the previous program is the following:

line	def	:	USE
1	$PTR1$	:	$\emptyset$
2	$PTR2$	:	$\emptyset$
3	$PTR3$	:	$PTR4$
4	OUT	:	$PTR5$

The references to array elements are converted to pointers, as mentioned earlier. Each pointer or array element occurrence has a unique identifier in the static DU. The instrumented code is:

```
    int t[5];
    remember("t", t);
1.  *dump("PTR1",&t[0])=1;
2.  *dump("PTR2",t+1)=2;
3.  *dump("PTR3",&t[2])=2*(*dump("PTR4",&t[1]));
4.  print(*dump("PTR5",&t[2]));
```

Assume that the array is placed at address 10. In this case the dynamic DU and the slice are:

step	line	def	:	USE	$DynSlice(def)$
1	1	10	:	$\emptyset$	$\emptyset$
2	2	11	:	$\emptyset$	$\emptyset$
3	3	12	:	$\{11\}$	$\{2\}$
4	4	OUT	:	$\{12\}$	$\{2,3\}$

3.3 Structures

The offset of the members of a structure could be determined statically but the computation of dynamic addresses would be quite complicated. Instead, the members of a structure will also be treated as pointers. In this way the structures are reduced to pointers. Consider the example:

```
    struct s {
        int a;
        int b;
    };
    struct s x,y;
1.  x.a=1;
2.  x.b=2;
3.  y=x;
4.  print(y.b);
```

Here, there are three structure members. The `x.a` in line 1, `x.b` in line 2 and `y.a` in line 4 are converted to $PTR1$, $PTR2$ and $PTR3$ respectively. Structures `x` and `y` are not converted to pointers, but are scalar variables. The static DU of the example is:

line	*def*	:	*USE*
1	$PTR1$	:	$\emptyset$
2	$PTR2$	:	$\emptyset$
3	y	:	$\{x\}$
4	OUT	:	$\{PTR3\}$

Let us suppose that the addresses of the structures and elements are the following:

item	x	x.a	x.b	y	y.a	y.b
address	01	01	02	03	03	04

As can be seen the address itself does not correctly describes a variable. Although the addresses of `x` and `x.a` are the same, they are still different. While `x.a` is located in a single memory cell, `x` occupies two cells: `01` and `02`. There is another reason for recording the structure size: the expression `y=x` in line 3. If the size of the structure is ignored the relation between `x.b` and `y.b` becomes indeterminable. In the instrumented code function `remember` has an additional parameter, the size of the variable. The addresses of the three structure members are written out by use of the function used for dump pointer values. The instrumented code is:

```
    struct s {
        int a;
        int b;
    };
    struct s x, y;
    remember("x", &x, sizeof(x));
    remember("y", &y, sizeof(y));
1.  *dump(&x.a)=1;
2.  *dump(&x.b)=2;
3.  y=x;
4.  print(*dump(&y.b));
```

Using the static DU and the dynamic information (addresses), the dynamic DU and the slice are the following:

step	line	def	:	USE	$DynSlice(def)$
1	1	01	:	$\emptyset$	$\emptyset$
2	2	02	:	$\emptyset$	$\emptyset$
3	3	03	:	{01}	{1}
		04	:	{02}	{2}
4	4	OUT	:	{04}	{2,3}

There are two memory locations defined in step 3, each with its own dependency. When the expression y=x is evaluated, the value of the structure x is copied into structure y. This means that two memory cells from the address 01 are copied into the cells starting at 03. The algorithm recognizes this fact and creates the two dependencies in the dynamic DU from the (single) dependency $y : x$ in the static DU.

3.4 Size of pointers

The method still hasn't been quite refined. As well as recording the size of a structure, the size of all variables and pointers must be known. It is obvious that a pointer can point to any structure or the element of an array can be a structure also. So the function dump has one additional parameter: the size of the pointed type. In this way the algorithm can compute the correct dynamic DU. It can find all addresses defined or used.

3.5 Same memory locations used during execution

During a program execution memory locations are allocated and released dynamically for some variables. It may happen that, after releasing such a memory location (which can be implicit or explicit), another variable gets the same address. Could this have some detrimental effects on the algorithm? If all variables were initialized before its first use, the answer is no. If the second variable is used without initialization, the algorithm uses the slice of the previous variable. This behaviour of the algorithm is also correct since it shows how the (probably bad) program works.

4 Handling unstructured statements

An issue which must be dealt with is how we should handle the jump statements in the dynamic slicing algorithm. In this section C-specific jump statements are considered, but the method could be used in other programming languages as well.

In the next part the handling of the goto statement is described, along with the `break`, `continue`, and `switch` statements.

4.1 The goto statement

Where a goto statement occurs, the D/U structure is built up as follows. First, so-called *"label variables"* are introduced. Let the defined variable (d) be the previously

introduced label variable called the real name of the label. It could also be an ordinal number, but for the sake of simplicity we use the previous name here. The use set (U) contains no "extra" variables, just the appropriate predicate variable, and we will find that it can contain label variables too.

The previously defined label variable is inserted into the use set (U) of those statements which occur after the corresponding label within the function. It is important to do this to the end of the function, not just in the appropriate block.

$i.$		def	:	USE
	`int i,j,k,l;`			
1.	`k=0;`	k	:	$\emptyset$
2.	`l=0;`	l	:	$\emptyset$
3.	`i=0;`	i	:	$\emptyset$
`l1:`				
4.	`j=0;`	j	:	$\{l1\}$
`l2:`				
5.	`k=k+i+j;`	k	:	$\{k,i,j,l1,l2\}$
6.	`l++;`	l	:	$\{l,l1,l2\}$
7.	`j++;`	j	:	$\{j,l1,l2\}$
8.	`if (j<2)`	$p8$	:	$\{j,l1,l2\}$
9.	`goto l2;`	$l2$	:	$\{p8,l1,l2\}$
10.	`i++;`	i	:	$\{i,l1,l2\}$
11.	`if (i<2)`	$p11$	:	$\{i,l1,l2\}$
12.	`goto l1;`	$l1$	:	$\{p11,l1,l2\}$
13.	`printf("%d",k);`	$o13$	:	$\{k,l1,l2\}$

Figure 5: Handling of the `goto` statement

If there are more labels, they are all handled in the same way. If the `goto` statement appears after the definition of the label, then of course it contains the just defined label variable. But this is not a problem because in the execution history it appears as a formerly defined variable. It can be defined by itself or by another `goto` statement. If no `goto` statement that jumps to a specific label is executed during the program, the last definition of that label remains undefined so it will not affect the result of the dynamic slice. The result contains all of the defined labels.

When the `goto` is executed during the program and the dynamic slice contains at least one of the statements after the definition of the label, then the result will at least contain the previous corresponding `goto` (and of course its predicate dependencies transitively). So it often unnecessarily increases the size of the dynamic slice and using lots of `goto` statements will make it hard to analyze the program.

An example is shown on Figure 5, and its results in Figure 6. As one might except, the use of `goto` statements resulted in a lot of dependencies.

Action (i^j)	*DynSlice*()	Action (i^j)	*DynSlice*()
1^1	$\emptyset$	16^{12}	$\{3,4,7,8,9,10,11\}$
2^2	$\emptyset$	17^4	$\{3,4,7,8,9,10,11,12\}$
3^3	$\emptyset$	18^5	$\{1,3,4,5,7,8,9,10,11,12\}$
4^4	$\emptyset$	19^6	$\{2,3,4,6,7,8,9,10,11,12\}$
5^5	$\{1,3,4\}$	20^7	$\{3,4,7,8,9,10,11,12\}$
6^6	$\{2\}$	21^8	$\{3,4,7,8,9,10,11,12\}$
7^7	$\{4\}$	22^9	$\{3,4,7,8,9,10,11,12\}$
8^8	$\{4,7\}$	23^5	$\{1,3,4,5,7,8,9,10,11,12\}$
9^9	$\{4,7,8\}$	24^6	$\{2,3,4,6,7,8,9,10,11,12\}$
10^5	$\{1,3,4,5,7,8,9\}$	25^7	$\{3,4,7,8,9,10,11,12\}$
11^6	$\{2,4,6,7,8,9\}$	26^8	$\{3,4,7,8,9,10,11,12\}$
12^7	$\{4,7,8,9\}$	27^{10}	$\{3,4,7,8,9,10,11,12\}$
13^8	$\{4,7,8,9\}$	28^{11}	$\{3,4,7,8,9,10,11,12\}$
14^{10}	$\{3,4,7,8,9\}$	29^{13}	$\{1,3,4,5,7,8,9,10,11,12\}$
15^{11}	$\{3,4,7,8,9,10\}$		

Figure 6: The result of program in Figure 5

4.2 The `break` statement

The `break` statement is practically equivalent to `goto` statement, which jumps out from the block of the appropriate `while`, `do...while`, `switch` or `for` statement to the first statement after this block. This statement can be handled as follows. The defined variable at every occurrence of the `break` statement should be an individual label variable. One form might be `break<Nr>`, where `<Nr>` is the ordinal number of the `break` statement within the program. All of the statements after the corresponding block are dependent on the previously defined label variable, just like in the case of `goto` statement. Note that if a label is placed just after the corresponding block and the `break` is replaced with a `goto` which jumps to that label, then the effect is the same.

An example of the `break` statement and results are shown in Figure 7 and Figure 8 respectively.

4.3 The `continue` statement

Like the `break` statement, we should define a separate label variable. This might be denoted by `continue<Nr>`, where `<Nr>` is the ordinal number of the `continue` statement within the program. It is defined in statements where `continue` occurs. The dependent statements are statements from the beginning of the block of the appropriate `for`, `while` or `do...while` statement to the end of the function. So the `continue` statement is always dependent upon itself.

$i.$		def	:	USE
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i\}$
5.	`b--;`	b	:	$\{p4, b\}$
6.	`i--;`	i	:	$\{p4, i\}$
7.	`if (b==0)`	$p7$	:	$\{b\}$
8.	`break;`	$break8$	:	$\{p7\}$
9.	`a++;`	a	:	$\{p4, a\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a, break8\}$

Figure 7: Handling of the `break` statement

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2, 3, 4\}$
6^6	$\{3, 4\}$
7^7	$\{2, 3, 4, 5\}$
8^8	$\{2, 3, 4, 5, 7\}$
9^{10}	$\{1, 2, 3, 4, 5, 7, 8\}$

Figure 8: The results of program in Figure 7

An example of the `continue` statement and results are shown in Figure 9 and Figure 10 respectively.

4.4 The `switch` statement

After the handling of `break` statement, the handling of the `switch` statement is quite straightforward.

At the place where the `switch` statement occurs a predicate variable is defined, just like in the case of `while` or `if`. All of the statements within the `switch` block are dependent on this predicate variable. If at least one statement within the switch block is included in the slice result, all of the `case` labels and the `default` label

$i.$		def	:	USE
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i, continue8\}$
5.	`b--;`	b	:	$\{p4, b, continue8\}$
6.	`i--;`	i	:	$\{p4, i, continue8\}$
7.	`if (b==0)`	$p7$	:	$\{b, continue8\}$
8.	`continue;`	$continue8$	:	$\{p7, continue8\}$
9.	`a++;`	a	:	$\{p4, a, continue8\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a, continue8\}$

Figure 9: Handling of the `continue` statement

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2,3,4\}$
6^6	$\{3,4\}$
7^7	$\{2,3,4,5\}$
8^8	$\{2,3,4,5,7\}$
9^4	$\{2,3,4,5,6,7,8\}$
10^5	$\{2,3,4,5,6,7,8\}$
11^6	$\{2,3,4,5,6,7,8\}$
12^7	$\{2,3,4,5,6,7,8\}$
13^9	$\{1,2,3,4,5,6,7,8\}$
14^4	$\{2,3,4,5,6,7,8\}$
15^{10}	$\{1,2,3,4,5,6,7,8,9\}$

Figure 10: The results of program in Figure 9

are included. Here the `break` statements are handled in the same way as described before.

An example of the `switch` statement and its results are shown in Figure 11 and Figure 12, respectively.

$i.$		def	:	USE
	`int a,b;`			
1.	`b=0;`	b	:	$\emptyset$
2.	`a=2;`	a	:	$\emptyset$
3.	`switch (a) {`	$p3$	:	$\{a\}$
	`case 1:`			
4.	`b=5;`	b	:	$\{p3\}$
5.	`break;`	$break5$	:	$\{p3\}$
	`case 2:`			
6.	`b=3;`	b	:	$\{p3\}$
	`case 3:`			
7.	`b++;`	b	:	$\{p3, b\}$
8.	`break;`	$break7$	:	$\{p3\}$
	`default:`			
9.	`b=6;`	b	:	$\{p3\}$
	`}`			
10.	`printf("%d",b);`	$o10$	:	$\{b, break5, break7\}$

Figure 11: Handling of the `switch` statement

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^6	$\{2,3\}$
5^7	$\{2,3,6\}$
6^8	$\{2,3\}$
7^{10}	$\{2,3,6,7,8\}$

Figure 12: The results of program in Figure 11

5 Experimental results

Several experimental results confirmed that our dynamic slices are more precise than the static one. Among the test sources there are 3 medium sized: the `bzip` (a compression utility), the `bc` (a scientific calculator) and the `less` (this is a powerful text viewer program). The sizes of these programs is shown in the following table.

prog	lines	executable	files	bytes	functions
bzip	4495	1595	1	130 458	73
bc	11555	3220	20	312 722	138
less	21489	5400	43	639 036	363

The first column is the name of the program, the second one means the total lines of the source, the third is the size of the executable code (i.e. without comments etc.), the fourth is the number of source files, the fifth is the total length of the source code in bytes, and the last one means the number of the functions within the program.

With help of our program we made several executions on several slice criteria for all the 3 sources. The number of the different slice criteria and the number of the executions are shown in the next table.

program	criteria	executions	coverage
bzip	154	18	68%
bc	57	49	63%
less	50	14	45%

The last column shows the coverage of the program. A statement is defined to be covered if at least once is executed during all the tests. The coverage means the percentage of the covered statements related to the whole program.

With a static slice generator tool (CodeSurfer, [10]) we made static slices, too. The results are shown in Figure 13.

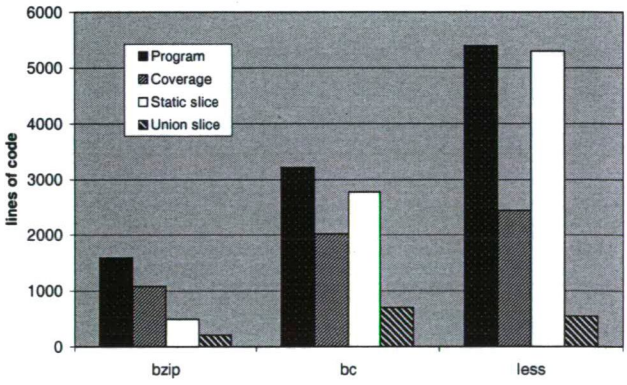


Figure 13: The average slice sizes

The first column shows the size of the executable code, the second the coverage, the third the average static slice (result of the CodeSurfer) and the last one is average of the so-called union slices generated by our dynamic slice generator tool. The union slice means the union of the all the generated slices (several executions + more results within one execution) to a certain statement.

6 Summary

Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be categorized according to whether they use static slicing or dynamic slicing methods. In applications such as debugging, the computation of dynamic slices is more preferable as it can produce more precise results.

There are several methods for dynamic slicing available in the literature, but most of them make use of the internal representation of the program execution with dynamic dependencies called the Dynamic Dependence Graph (DDG). A big drawback of these methods is that the size of the DDGs is unbounded, because it includes a distinct vertex for each occurrence of a statement.

In [9] a new forward global method for computing dynamic slices of C programs was introduced. The algorithm determines the dynamic slices for any program instruction, in parallel with program execution, but it was worked out only for a simple program language.

To make the method usable for real programs, many problems had to be solved. This paper focused on two of them: the handling of pointers and unstructured jump statements. A method for handling the pointers, arrays, structures, `goto`, `break`, `continue` and `switch` statements of the `C` programming language was described, as well.

The main advantage of our algorithm is that it can be applied to real size `C` programs as its memory requirements are proportional to the number of different memory locations used by the program (which is in most cases much smaller than the size of the execution history—which is, actually, the absolute upper bound).

We have already developed a program where we implemented the forward dynamic slicing algorithm for C language programs. According to our preliminary trials, the memory requirements of the algorithm is indeed proportional to the number of different memory locations used by the program, which is much less than the size of the execution history.

References

[1] Agrawal, H., DeMillo, R. A., and Spafford, E. H. Debugging with dynamic slicing and backtracking. *Software—Practice And Experience*, 23(6):589-616, June 1993.

[2] Beck, J., and Eichmann, D. Program and Interface Slicing for Reverse Engineering. In *Proc. 15th Int. Conference on Software Engineering*, Baltimore, Maryland, 1993. IEEE Computer Society Press, 1993, 509-518.

[3] Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimóthy, T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303-322.

[4] Gallagher, K. B., and Lyle, J. R. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering* 17, 8, 1991, 751-761.

[5] Gyimóthy, T., Beszédes, Á., and Forgács, I. An Efficient Relevant Slicing Method for Debugging. In *Proc. 7th European Software Engineering Conference (ESEC)*, Toulouse, France, Sept. 1999. LNCS 1687, pages 303-321.

[6] Korel, B., and Rilling, J. Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*, Linkoping, Sweden, May 1997.

[7] Rothermer, G., and Harrold, M. J. Selecting tests and identifying test coverage requirements for modified software. In *Proc. ISSTA'94* Seattle. 1994, 169-183

[8] Weiser M. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4, 1984, 352-357.

[9] Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J., and Gyimóthy T. *Dynamic Slicing Method for Maintenance of Large C Programs.* At the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001). Lisbon, Portugal, March 14-16, 2001.

[10] Homepage of `CodeSurfer`: `http://www.grammatech.com/products/codesurfer/`

Acta Cybernetica 15 (2002) 509–528.

XML Semantics Extension

Ferenc Havasi*

Abstract

Nowadays one of the best standards for storing structured data is XML. The key idea behind our paper is based on a relationship between XML documents and attribute grammars. This parallel makes it possible to apply techniques of attribute grammars (semantics rules) to the XML environment.

We present a new method for extending XML. After a background discussion we formally introduce a way of defining real XML attributes using semantics rules. This allows us to incorporate semantics rules into XML files in such a way that it does not violate the original XML specification and is suitable for compressing using learning. After learning an associated semantics rule file preserves the relationship between attributes.

Introduction

These days one of the most popular standards in use for storing structured information is XML. More and more applications are able to export data in an XML format, more databases are stored in XML, and XML processing techniques are becoming more generic. If this trend continues, XML will eventually be present in almost every part of the informatics sphere. Because of this the new research results related to XML should prove important the future.

The main idea behind our paper is based on a connection between XML documents and attribute grammars. The analogy makes it possible to apply techniques of attribute grammar (semantics rules) to the XML environment. The first notion of including semantics to XML was published in [9], but here we shall introduce a new approach. We create a format which makes it possible for us to define a real XML attribute via semantics rules. The new set of semantics rules then become an organic part of XML documents and do not violate the original XML specification.

In the first section we review the fundamentals of XML and attribute grammars. Next, we throw light on the relationship between them, show how to extend XML documents with semantics rules, and then introduce a new (SRML) format to describe it. In Section 3 we discuss new S-SRML and L-SRML descriptions, which are counterparts of S-attributed and L-attributed grammars. The next section deals with the learning of SRML descriptions. In Section 5 we discuss the

*Research Group of Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544145, email: havasi@rgai.hu

implementation, and present experimental results. Finally we briefly elaborate on related and future work, and give a summary of our conclusions.

1 Background

To facilitate an understanding of this paper we first give a brief overview of the necessary fundamentals such as XML technology, formal and attribute grammars. Rather than give a detailed account of the above, we will just present the most important parts using an example.

1.1 XML

In this section we provide a brief introduction to the basics of XML [2], which we will make use of later.

1.1.1 Document:

An XML document has an html like text-based format. Its components are called *elements*. An *element* always begins with a *start-tag* and ends with an *end-tag*. Take, for instance,

```
<section>A long text</section>
```

An element may contain other elements and/or text or it may be empty. With the start-tag of an element it is possible to define *attributes*, for example,

```
<section title="Introduction">A long text</section>
```

In Figure 1 we have a more complicated example using a numeric expression.

1.1.2 DTD:

This is a file containing a meta language. Its description makes it possible for us to define the structure of an XML document. This language is called *DTD*. We can specify the content of an element, which other element or text can be inserted, and in which order. In our case the following two meta-tags are the most important:

!element: This tag specifies a regular expression[1]. It defines the element and the order it assumes.

!attlist: This tag specifies the type and allowed values of the attributes of an element.

Figure 2 shows one such DTD file:

1. The element num can contain only text (#PCDATA), and has a required (#REQUIRED) attribute called type, its value being real or integer-valued.

[1] It is possible to transform this expression to EBNF format [9].

```
<expr>
  <multexpr op="mul" type="real">
    <expr type="int">
      <num type="int">3</num>
    </expr>
    <expr type="real">
      <addexpr op="add" type="real">
        <expr type="real">
          <num type="real">2.5</num>
        </expr>
        <expr type="int">
           <num type="int">4</num>
        </expr>
      </addexpr>
    </expr>
  </multexpr>
</expr>
```

Figure 1: A possible XML form of the expression 3*(2.5+4).

```
<!ELEMENT num (#PCDATA) >
<!ATTLIST num type ( real | int ) #REQUIRED
>
<!ELEMENT expr ( num |  multexpr  |  addexpr ) >
<!ATTLIST expr type ( real | int ) #IMPLIED
>
<!ELEMENT multexpr ( expr , expr ) >
<!ATTLIST multexpr op  ( mul | div )  #REQUIRED
                  type ( real | int ) #IMPLIED
>
<!ELEMENT addexpr ( expr , expr ) >
<!ATTLIST addexpr op  ( add | sub )  #REQUIRED
                 type ( real | int ) #IMPLIED
>
```

Figure 2: The DTD of the corresponding XML file for the previous figure.

2. The expr element contains a num or a multexpr or an addexpr element (| mark means or), and there is an optional (#IMPLIED) attribute called type.
3. The elements multexpr and addexpr must contain two expr elements, and have two attributes: a necessary one called op and an optional one called type.

1.2 Parsing XML

Basically there are two kinds of interfaces of XML parsers:

SAX interface: This interface is very simple: all information is passed from the parser to the application via function calls. For example if there is a *start-tag* the parser generates a startTag(element_name)-like call.

DOM tree interface: Using this method the parser builds a tree called a DOM tree. Every element and attribute is represented as a vertex. The root node of the tree is the root element of the document. The edges represent incorporated dependences. A corresponding tree of the example in Figure 1 is shown in Figure 3.

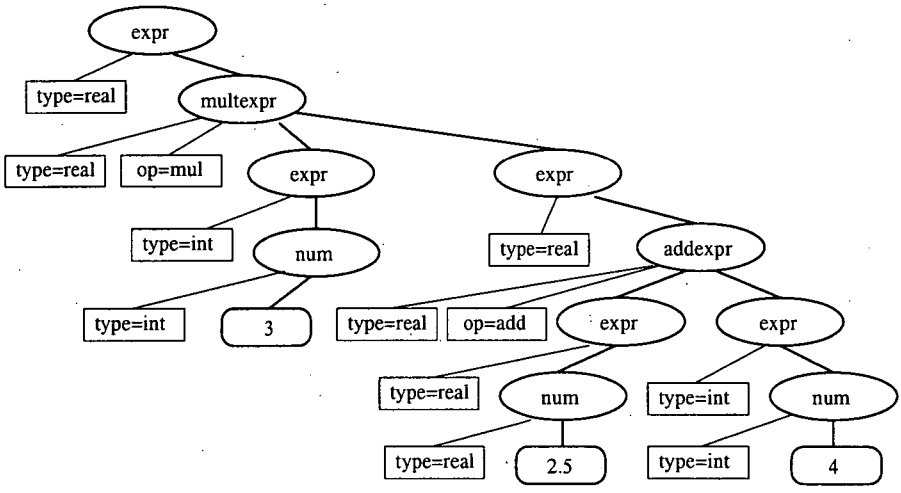


Figure 3: DOM tree of example in Figure 1.

1.3 Formal languages and derivation trees

A general way of specifying a syntax is by using a formal grammar. A formal grammar is a G=(N,T,S,P) set, where N is the set of the nonterminal symbols, T is the one of the terminal symbols, S is the start-symbol and P is a set of transformation rules. Take, for example,

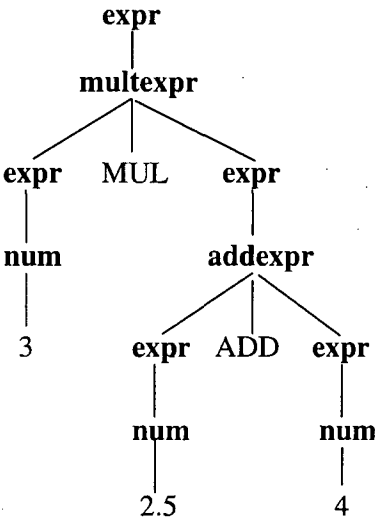
```
N = { expr, multexpr, addexpr, num }
S = expr
T = { "ADD" , "MUL" , NUM }
P :
  expr     -> num | multexpr | addexpr
  addexpr  -> expr "ADD" expr | expr "SUB" expr
```

```
multexpr -> expr "MUL" expr | expr "DIV" expr
num      -> NUM
```

The grammar in it specifies the format of simple numeric expressions. At the left side of every rule there is only one nonterminal, so it is a context free (CF) grammar. The derivation process starts at *expr*. It may be a simple number (*num*), a multiplicative or an additive expression (*multexpr* or *addexpr*). The *num* nonterminal is a simple number token (NUM), whereas the multiplicative expression contains two expressions and a MUL or DIV token is placed between them. The additive expression also consists of two expressions with an ADD or SUB token between them.

For any given input word a derivation tree can be drawn. In the root of a derivation tree there is the start-symbol (expr), and below nonterminals there are substituted expressions. If we concatenate the leaves we get the input word.

If this input is 3 MUL (2.5 ADD 4), the derivation tree looks like the following:



1.4 Attribute grammars

An attribute grammar [7] contains a CF grammar, and

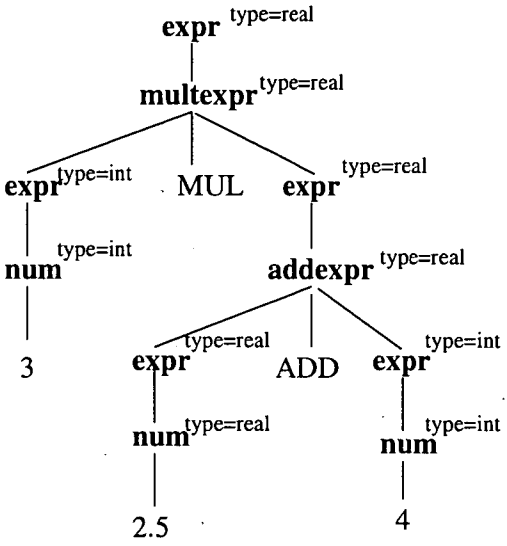
attributes: We can assign attributes to *nonterminals*. For example, by assigning the x attribute to the S nonterminal we get the S.x attribute. In general there are two types of attributes: inherited and synthesized.

semantics rules: We can assign semantics rules for each *formal rule*. The semantics rules define a formula for computing the value of an attribute. For example:

```
S -> A B
  S.x = A.x + B.x
```

In this semantics rule the attribute x of S can be calculated from the attribute x of A and B.

If we supplement the derivation tree with the values of attribute occurrences we get an attributed derivation tree. All attribute occurrences are calculated once and only once. Take, for instance the attributed derivation tree of the expression example:



2 XML semantics

2.1 Relationship between attribute grammars and XML

Figure 3 shows the DOM tree of the previous XML document example. If we recall the attributed derivation tree, we will recognize that an XML document can be viewed as an attributed derivation tree. Hence we expect to see the following analogy:

Attribute Grammars	XML
nonterminal	element
formal rules	element specification in DTD
attribute specification	attribute specification in DTD
semantic functions	???

There is one key concept in AG that has no counterpart in the XML environment: semantic functions. This is a very important concept in AG and there are a lot of techniques based on it. It might be useful to apply these techniques to the XML environment.

In an XML document the values of all attribute occurrences are stored in a direct form. If we were able to define semantics rules for XML attributes it would be sufficient to store these rules once and, making use of them, the concrete value of the attributes could be calculated. Then it would not be necessary to store them in the document.

So our idea is to define XML attributes using semantics rules. Our definition of semantics rules will be an organic part of XML document.

2.2 Complete/Reduce

We will define only IMPLIED XML attributes with semantics rules. In an XML document this kind of attribute is not required, so the DTD of the document validates the XML document whether or not these attributes are defined.

Now let us consider the following figure:

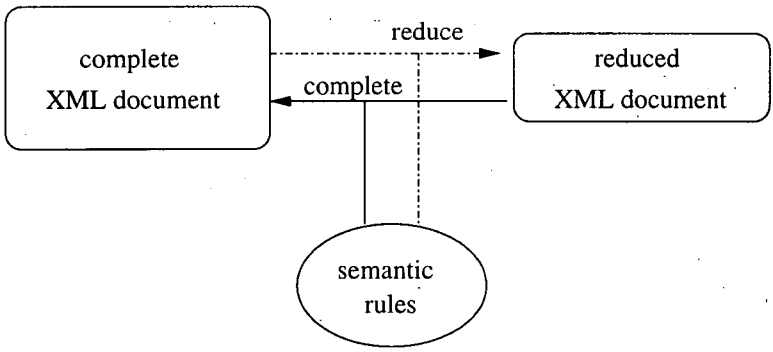


Figure 4: The complete and reduce methods.

In this diagram we notice that the *complete* method creates a complete XML document from a reduced one. All nondefined IMPLIED attributes that have a semantics rule will be calculated using this rule.

In contrast, the *reduce* method does the opposite of the previous one. The input of it is a complete XML document, along with some semantics rules. All IMPLIED attributes which have a *correct*[2] semantics rule will be deleted from the document, so the output will be a reduced version of the input XML document.

2.3 Specifying semantics rules

We define a meta language called SRML (Semantics Rule Meta Language) to describe semantics rules, which has an XML based format. The corresponding DTD of this language is the following:

[2]Note that we can use rules even if they are not absolutely correct rules. If the result of the semantics rule differs from the actual attribute value it will be kept. So the *reduce* method is really the inverse of the *complete* method.

```
<!ELEMENT semantic-rules ( rules-for* ) >
<!ELEMENT rules-for ( rule* ) >
<!ATTLIST rules-for root  NMTOKEN  #REQUIRED>
<!ELEMENT rule ( expr ) >
<!ATTLIST rule element  NMTOKEN  #REQUIRED
               attrib  NMTOKEN  #REQUIRED
>
<!ELEMENT expr ( binary-op | attribute | data |
                 no-data | if-element | if-expr |
                 if-all | if-any | current-attribute |
                 position | external-function ) >
<!ELEMENT binary-op ( expr, expr) >
<!ATTLIST binary-op op  (add | sub | mul | div | exp | equal |
                         not-equal | less | greater | or |
                         xor | and | nor | contains | concat |
                         begins-with | ends-with )    #REQUIRED
>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute element NMTOKEN                    "srml:this"
                    num     NMTOKEN                    "0"
                    from    ( begin | current | end ) "current"
                    attrib  NMTOKEN                    #REQUIRED
>
<!ELEMENT if-element ( expr, expr)>
<!ATTLIST if-element from ( begin | end ) "begin">
<!ELEMENT position EMPTY>
<!ATTLIST position element  NMTOKEN                    "srml:all"
                   from     ( begin | end )            "begin"
>
<!ELEMENT if-all ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-all element    NMTOKEN                    "srml:all"
                attrib      NMTOKEN                    "srml:all"
>
<!ELEMENT if-any ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-any element    NMTOKEN                    "srml:all"
                attrib      NMTOKEN                    "srml:all"
>
<!ELEMENT current-attribute EMPTY>
<!ELEMENT if-expr (expr , expr , expr ) >
                                  <!-- condition , if, else -->
<!ELEMENT data (#PCDATA) >
<!ELEMENT no-data EMPTY>
<!ELEMENT extern-function (param)*>
<!ATTLIST extern-function name NMTOKEN #REQUIRED>
<!ELEMENT param (expr)>
```

A DTD can be viewed as a formal grammar [9]: elements will be nonterminals and the descriptions of the element will be formal rules. We would like to define some semantics rules for these formal rules.

The meaning of the elements of SRML are:

semantics-rules : This is the root element of SRML.

rules-for : This element gathers together the semantics rules of a formal rule. In a DTD there is only one working description of an element so the formal rule is determined by that element, which is on the left side of the formal rule. This is the *root* attribute of the *rules-for* element.

rule : This element describes a semantics rule. We have to specify which *attribute* of which *element* we are going to define in this semantics rule and its value in *expr*. If the value of the *element* is "*srml:root*" then we define an attribute of the root element.

expr : An expression can be a binary-expression (*binary-op*), an *attribute*, a directly defined value (*data* or *no-data*), a conditional expression (*if-expr*,*if-all* or *if-any*), a syntax-condition (*if-element* and *position*) or an external function call (*extern-function*).

if-element : In a DTD element description one can specify a regular expression (with maybe +,*,?,... marks). This element provides us with the possibility of testing the actual form of the input. It contains two *expr* elements. As an expression the value of the *if-element* is true or false depending on the following: the name of the first *expr*th child (element) in the actual rule equals to the value of the second *expr*. The *from* attribute can specify which direction to operate. In other words it is possible to take an index from the end of the level without actually knowing how many children the parent has.

binary-op : This element is an ordinary binary expression.

position : Returns a 0 based index which identifies the current element's position taking into consideration the *element* attribute. Possible directions are as follows : begin, end. It is possible to use the *srml:all* identifier, in which case the index returned will be the actual overall position in the DOM tree level. If an *element* name is specified then the returned index will be n where the element has $n-1$ predecessors or successors with the same element name.

attribute : The attribute is determined by its *element*, *attrib*, *from* and *num* attributes. In the actual rule this is the *num*th element where the name matches the value of the *element* (if it is "*srml:any*" it can be anything, if it is "*srml:root*" then it is an attribute of the root) *from* the *begin*ning, the *current* element or the *end*ing. If the attribute does not exist it will be handled as *no-data*.

if-expr : This is an ordinary conditional expression. The first *expr*ession is the condition and depends on whether if it is true (not zero) or not. The value of the *if-expr* will be the value of the second or third *expr*ession.

if-all : This is an iterated version of the previous *if-expr* expression. The first *expr* is computed for all matching attributes (each selected by *element* and *attrib*ute, which can take a concrete value or "*srml:all*)". We can refer to the value of this attribute using the element *current-attribute*. If the condition (first *expr*) is true for all matching attributes, the value of it is the value of the second *expr* otherwise it is the third *expr*.

if-any : This is almost the same as the previous one except that it is sufficient that the condition be true for at least one matching attribute.

current-attribute : This is the loop variable of *if-any* and *if-all* elements.

data : This element has no attribute and usually contains a number or a string.

no-data : This element means that this attribute cannot be computed – it is often present in some branches of conditional expressions.

extern-function : This element makes an external function call handled by the implementation. It makes SRML more easily extendable.

param : This describes a parameter of an *external-function*.

A valid SRML description must be *consistent*. This means there mustn't be any attribute occurrences that are defined more than once.

2.4 An SRML example

Here are some interesting portions of the SRML description from the previous numeric expression example.

```
<semantic-rules>
<rules-for root="expr">
<rule element="srml:root" attrib="type">
  <expr>
    <attribute element="srml:any" num=1 attrib="type" from="begin"/>
  </expr>
</rule>
</rules-for>
<rules-for root="addexpr">
<rule element="srml:root" attrib="type">
  <expr>
    <if-expr>
    <expr>
      <binary-op op=or>
```

```
        <expr>
        <binary-op op=equal>
          <expr>
            <attribute element="expr" num=1 attrib="type"
                                            from="begin"/>
          </expr>
          <expr><data>real</data></expr>
        </binary-op>
        </expr>
        <expr>
        <binary-op op=equal>
          <expr>
            <attribute element="expr" num=2 attrib="type"
                                            from="begin"/>
          </expr>
        <expr><data>real</data></expr>
        </binary-op>
        </expr>
      </binary-op>
    </expr>
    <expr><data>real</data></expr>
    <expr><data>int</data></expr>
    </if-expr>
  </expr>
</rule>
</rules-for>
</semantic-rules>
```

The set of rules for *multexpr* is very similar to that for *addexpr*.

3 Attribute grammar types – SRML description types

The attribute grammars can be classified according to the evaluation method employed [1]. There are S-attributed grammars, L-attributed grammars, OAG attributed [6] grammars and so on.

By analogy we could introduce S-, L-, ASE-,... SRML descriptions. Here we only define S- and L-SRML descriptions.

Actually, there are only two relevant factors which we need to know in the SRML description to decide whether it is an S/L-SRML description. The first one is the set of defined attributes associated with the rules, while the second is the set of referenced attributes in these definitions.

3.1 The S-SRML description

S attributed grammars are the simplest attribute grammars: they have only synthesized attributes. As in XML, in SRML we do not distinguish between synthesized attributes and inherited ones. In this environment we can define an S-SRML description, in analogy with S-attributed grammars:

definable attributes : In each rule we can only define the attributes of the (srml:)root nonterminal (the element which is on the left side), because synthesized attributes are only definable in a rule if the root element contains them.

usable attributes in definitions : All attributes in this rule presume that there are no circular dependencies.

3.2 The L-SRML description

An L-attributed grammar can contain synthesized and inherited attributes, but the dependencies between them must be evaluated in one left-to-right pass. In the SRML environment it means the following:

definable attributes : All available attribute occurrences bearing consistency (see in 2.3) in mind.

usable attributes in definitions : We use one left-to-right pass to evaluate the attributes. In a rule environment we first calculate the attributes of the children nonterminals, and after the attributes of the root nonterminal in a suitable order. An SRML description is called an L-SRML description if there is a suitable order in attributes which carries out the following: if there is an attribute reference in the definition of an attribute then the value of referenced attribute has already been calculated.
To be more precise:

- In the definition of an attribute of the root there can be attributes of any children, or those attributes of the root that have been defined earlier in the SRML description.
- In the definition of an attribute of a child there can be attributes of any children which lie to the left side of it, or are those attributes of the same child that have been defined earlier in the SRML description.

4 Learning of the SRML description

A relatively large XML document usually contains lots of redundancies. This means that many attributes can be computed from other attributes. However, in many cases the computation rules are not trivial and the recognition of these may require

machine learning approaches [8]. The learning of the SRML description means detecting relationships between XML attributes. These relationships can be described in an SRML format. This method has two obvious applications, namely:

compressing: After learning an SRML description we can then the apply the *reduce* algorithm. This reduced version of the document is usually much smaller than the original one. From it and the SRML description the original document is recoverable, so it can be regarded as a form of data compression[3].

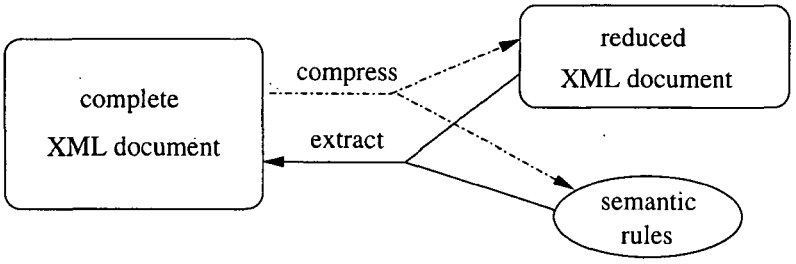


Figure 5: Using the learning of an SRML description to (de)compress XML files.

understanding: The SRML description provides us with hitherto unknown correspondences. When we store a database in XML document format we can find correspondences in it as well.

The learning method can be fault-tolerant. In this case the detected correspondences may be better (the data may have measuring errors). If we only need estimated values and the (XML) database is very large, we can use the short SRML description.

In [5] and [12] a machine learning method was introduced to infer the semantic rules of attribute grammars. There is a close relationship between attribute grammars and SRML, hence this learning approach can be used to produce SRML description from XML documents.

5 Implementation

The structure of the implementation is given in Figure 6. The implementation of a complete version of this tool is under way (in a JAVA and DOM environment). Here we present results using a special version of this tool. This tool's task is to read CPPML files and reduce them.

[3]Of course after this method we can use other ordinary compression methods like those mentioned in [10].

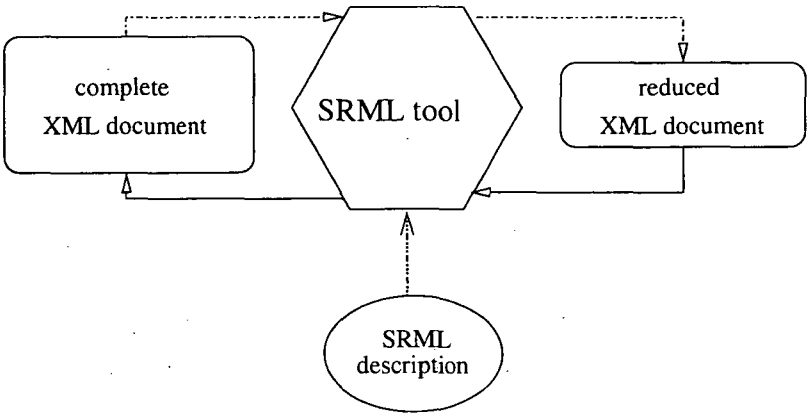


Figure 6: The structure of the implementation.

5.1 CPPML

CPPML (C++ Markup Language) [3] is a markup language for describing the structure of programs written in C++. This may be generated using a Columbus reverse engineering tool [4] for any C++ program.

Here is a brief extract of a C++ program:

```
class _guard : public std::map<std::string, _guard_info>
  {
    public:
        void registerConstruction(const type_info & ti)
    { (*this)[ti.name()]++ ;
    }

    void registerDestruction(const type_info & ti)
    { (*this)[ti.name()]-- ;
    }

    void dumpInstances(const char * file, bool bAppend)
    {
      fstream f(file, bAppend  ?  ios_base::out|ios_base::app
                               :  ios_base::out) ;
      iterator i ;
      for(i=begin(); i!=end(); i++)
        f  << i->second._count << "  -  " << i->second._max_count
           << "  :  "  << i->first<< endl ;
      f << endl;
    }
  } ;
```

```
} ;
```

The CPPML representation of the above could be like the following:

```
...
<class id="id20097" name="_guard"
    path="D:\CAN_Test\SymbolTable\Input\CANGuard.h" line="71"
    end-line="90" visibility="global" abstract="no"
    defined="yes" template="no" template-instance="no"
    class-type="class">
  <function id="id20102" name="registerConstruction"
      path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      line="75" end-line="76" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      body-line="75" body-end-line="76">
    <return-type>void</return-type>
    <parameter id="id20106" name="ti"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="74" end-line="74" const="yes">
      <type>type_info&amp;amp;</type>
    </parameter>
  </function>
  <function id="id20109" name="registerDestruction"
      path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      line="79" end-line="80" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      body-line="79" body-end-line="80">
    <return-type>void</return-type>
    <parameter id="id20113" name="ti"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="78" end-line="78" const="yes">
      <type>type_info&amp;amp;</type>
    </parameter>
  </function>
  <function id="id20116" name="dumpInstances"
      path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      line="83" end-line="89" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      body-line="83" body-end-line="89">
    <return-type>void</return-type>
    <parameter id="id20120" name="file"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="82" end-line="82" const="yes">
```

```
      <type>char*</type>
    </parameter>
    <parameter id="id20122" name="bAppend"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="82" end-line="82" const="no">
      <type>bool</type>
    </parameter>
  </function>
</class>
...
```

5.2 A real example: reducing of CPPML

As you might have noticed in a CPPML description many attributes can be calculated or estimated using other attributes.

For example, the *kind* of a *function* is a *constructor* if the *name* of the *function* is equal to the name of the *class*. We can describe it in SRML form:

```
<rules-for root="class">
  <rule element="function" attrib="kind">
    <expr>
      <if-expr>
        <expr>
          <binary-op op="equal">
            <expr>
              <attribute attrib="name"/>
            </expr>
            <expr>
              <attribute attrib="name" element="srml:root"/>
            </expr>
          </binary-op>
        </expr>
        <expr>
          <data>constructor</data>
        </expr>
        <expr>
          ...
        </expr>
      </if-expr>
    </expr>
  </rule>
</rules-for>
```

Actually we can not only describe valid rules here, but also make estimations (the *reduce* method will only delete the matching attributes). Let us look at some types of estimations:

1. the declaration of a function or variable starts and ends on the same line.
2. the implementation of the functions of a class are usually in the same file.
3. the parameters of a function are also in the same file, perhaps on the same line.

Here are some parts of the corresponding SRML description:

```
<rules-for root="function">
  <rule element="parameter" attrib="end-line">
    <expr>
      <attribute attrib="line"/>
    </expr>
  </rule>
  <rule element="parameter" attrib="line">
    <expr>
      <attribute attrib="line" num="-1"/>
    </expr>
  </rule>
  <rule element="parameter" attrib="path">
    <expr>
      <attribute attrib="path" num="-1"/>
    </expr>
  </rule>
</rules-for>
```

The full description of our SRML description for CPPML can be found at the following internet site: `http://xml.rgai.hu/`.
Let us illustrate the previous part of the CPPML description using the procedure *reduce*:

```
<class id="id20097" name="_guard"
    path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
    line="71" end-line="90" visibility="global" abstract="no"
    defined="yes" template="no" template-instance="no"
    class-type="class">
  <function id="id20102" name="registerConstruction"
      line="75" end-line="76" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h">
    <return-type>void</return-type>
    <parameter id="id20106" name="ti" line="74" const="yes"
        ellipsis="no">
      <type>type_info&amp;amp;</type>
    </parameter>
  </function>
  <function id="id20109" name="registerDestruction" line="79"
```

```
      end-line="80" const="no" virtual="no" pure-virtual="no"
      kind="normal" visibility="global">
    <return-type>void</return-type>
    <parameter id="id20113" name="ti" line="78" const="yes"
       ellipsis="no">
      <type>type_info&amp;amp;</type>
    </parameter>
  </function>
  <function id="id20116" name="dumpInstances" line="83"
      end-line="89" const="no" virtual="no" pure-virtual="no"
      kind="normal" visibility="global">
    <return-type>void</return-type>
    <parameter id="id20120" name="file" line="82" const="yes"
        ellipsis="no">
      <type>char*</type>
    </parameter>
    <parameter id="id20122" name="bAppend" const="no"
       ellipsis="no">
      <type>bool</type>
    </parameter>
  </function>
</class>
```

5.3 Experimental results

The results of applying the *reduce* method are listed in Figure 7. The column "Orig Size" is the size of the original CPPML file, and "New Size" is its size after the reduction. The original CPPML file contained "Attribute" attribute items, and we eliminated "Deleted" ones because it was possible to compute their values using the rules. In the "Not matched" column there is the attribute number which is computable from the rules but its value is not correct. Here we do not delete these attributes.

Prog Name	Orig Size	New Size	Attributes	Deleted	Not matched
AppWiz	3589076	2192377	115684	43451	10151
jikes	2257728	1745720	93045	26100	9964
leda	11673855	9023687	405916	88322	26032

Figure 7: Results of CPPML reduction.

The size of this CPPML SRML description is less than 4 kilobytes, and from the reduced version of the document and this SRML description the original one can be recovered, so the *reduce* method is a form of data compression. After this "compression" we can apply some ordinary compression technique. The density of

information in the *reduced* document is higher than in the original document, so it would be interesting to examine the compression ratio using a zip-like compression.

We compressed the original and the *reduced* document using gzip, and the results are shown in Figure 8. The difference between the compression ratio using gzip on the original and the *reduced* documents is about 1%.

Prog Name	Orig Size	Gziped orig	Orig ratio	New Size	Gzipped new	New ratio
AppWiz	3589076	244659	0.068	2192377	167749	0.076
jikes	2257728	177223	0.078	1745720	140681	0.081
leda	11673855	821074	0.070	9023687	709922	0.079

Figure 8: Results of compressing the original and the reduced CPPML description.

6 Note on a related study

The first motion of adding semantics to XML was published in [9]. After a brief introduction to XML that paper provides a method for transforming the element description of DTD into EBNF formal rule description.

Afterwards it introduces its own SRD (Semantics Rule Definition) composed of two parts: the first one describes the semantics attributes[4], while the second one gives a description of how to compute them. SRD is also XML-based.

The main difference between the approach outlined in this article and ours is that we provide semantics rules not just for newly defined attributes but also for real XML ones. Our approach makes the SRML description an organic part of XML documents. As the defined attributes are IMPLIED, either the complete or the reduced document is validated by the original DTD. This kind of semantics definition could offer a useful extension for XML techniques.

Our SRML description also differs from the SRD description in that article. In SRD the attribute definition of elements with a + or * sign is defined in a different way from the ordinary attributes definition and can only reference the attributes of the previous and subsequent element. The references in our SRML description are more generic, and all expressions are XML-based.

7 Conclusion and Future Work

Our method defines correspondences between XML attributes and stores them in an SRML format. It is not necessary to store attributes which are computable with the SRML description - this is the reduced version of the XML document. The complete document is recoverable from the reduced version and the SRML description.

[4]These are newly defined attributes which differ from those in XML files.

We can use learning to create an SRML description from an XML document. It can supply us with a description of valid correspondences between attributes. We can utilize it to compress the document because the sum total of the given SRML description and the reduced version of the document is generally much smaller than that for the entire document.

We are developing a tool which will be able to handle a general SRML description and use both the *complete* and *reduce* methods.

The theory of learning SRML descriptions is an interesting and open area of research. In future experiments we plan to extend our tool with various learning modules so as learn SRML descriptions from any XML documents, keeping on eye on the size of the reduced version of documents.

References

[1] H. Alblas, *Introduction to attribute grammars*, Springer Verlag, In Proc. of SAGA (H. Alblas and B.Melichar eds.) LNCS **545** (1991), 1–16.

[2] T. Bray, J. Paoli, and C. Sperberg-MacQueen, *Extendable markup language*, 1998.

[3] R. Ferenc, *CPPML - An Implementation of the Columbus Schema for C++.*

[4] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen, *Tool for reverse engineering large object oriented software*, SPLST (2001), 16–27.

[5] T. Gyimóthy and T. Horváth, *Learning semantic functions of attribute grammars*, Nordic Journal of Computing **4** (1997), no. 3, 287–302.

[6] U. Kastens, *Oredered attribute grammars*, Acta Informatica **13** (1980), 229–256.

[7] D. E. Knuth, *Semantics of context-free languages*, vol. 2, pp. 127–145, Springer-Verlag, New York, 1968.

[8] T. Mitchell, *Machine learning*, McGraw-Hill, 1997.

[9] G. Psaila and S. Crespi-Reghizzi, *Adding Semantics to XML*, Second Workshop on Attribute Grammars and their Applications, WAGA'99 (Amsterdam, The Netherlands) (D. Parigot and M. Mernik, eds.), INRIA rocquencourt, 1999, pp. 113–132.

[10] XML Compression Tools, *http://sourceforge.net/projects/xmlppm/.*

[11] XML parsers, *XML-sofware. http://www.xmlsoftware.com/parsers.html.*

[12] Sz. Zvada and T. Gyimóthy, *Using decision trees to infer semantics functions of attribute grammars*, Acta Cybernetica **15** (2001), 279–304.

Acta Cybernetica **15** (2002) 529–545.

Mathematical models for simulation of continuous grinding process with recirculation

Piroska B. Kis*, Csaba Mihálykó†, and Béla G. Lakatos‡

Abstract

New mathematical and computer models and simulation programs were elaborated for studying processes of continuous grinding mills working with classification and partial recirculation of the product. The computer models were developed on the basis of the axial dispersion model taking into consideration also the effects of the mixing of the material to be ground. The effects of changes of parameters of both the mill and the material were studied. The stationary states of the continuous grinding mills working with and without classification and recirculation were compared to each other. The mathematical models and the computer programs developed are suitable for computing the processes of the grinding mills either with or without recirculation. They are usable for simulation based analysis and design of both continuous and batch grinding devices.

1 Introduction

Grinding is an important technological process in process industry. The two basic types of grinding are the batch and the continuous grinding. The mathematical analysis of batch grinding has been carried out in a number of aspects [1-9]. Considerable experimental research has been taken on continuous grinding [10-13], too, but fewer results were published concerning the mathematical modelling of the continuous grinding processes.

The mathematical description of continuous grinding mills can be formulated by means of distributed parameter models. One of these types of models, derived from the integro-differential equation of continuous grinding, was published and verified by Mihálykó *et al.* [13]. By using this discrete type model, the effects of system parameters on the behaviour and performance of the mills can be studied extensively. However, the classification and partial recirculation of the material to be ground, that is an important and often used solution for increasing the quality of the product of griders, was not included into this model.

*Institute of Natural Sciences, College of Dunaújváros, e-mail: piros@mail.poliod.hu

†Department of Mathematics and Computing, University of Veszprém, e-mail: mihalyko@almos.vein.hu

‡Department of Process Engineering, University of Veszprém, e-mail: lakatos@fmt.vein.hu

The aim of the present paper is to develop a generalised model, taking into consideration the classification and recirculation processes of the material to be ground. The kinetics of grinding is modelled by the fundamental grinding equation. The computer models presented are based on the resulting partial integro-differential equation. The newly developed computer models are suitable for simulation of continuous grinding mills working with and without classification and recirculation.

2 Mathematical model of continuous grinding with recirculation

In order to review the theory let us introduce the following notation. Let t denote time, t_r - the time spent by material in the recycling loop, X - length of the mill, x - axial coordinate of the mill, L - particle size, $\nu(x,t)$ - the average linear velocity of the particles in the mill, $D(x,t)$ - the axial dispersion coefficient characterising the mixing of particles in the mill, and $m(x,L,t)$ - the mass density function of particles in the mill. The mass density function characterises the size distribution of particles by means of which $m(x,L,t)\mathrm{d}L$ expresses the mass of particles at axial coordinate x of the mill at time of t within the particle size interval (L, $L+\mathrm{d}L$) in a unit volume of the mill. Let $m_{in}(L,t)$ denote the mass density function of the particles entering the mixing device, $m_r(L,t_r)$ the mass density function of the particles leaving the classifier and entering the mixing device again.

Let $f(L,t)$ denote the mass flow density function of the particles leaving the mill, as it shown in Fig.A, $f_f(L,t)$ denotes the mass flow density function of the particles entering the mill, and $f_{out}(L,t)$ is the mass flow density function of the particles leaving the grinding system. Let $f_r(L,t)$ denote the mass flow density function of the particles classified and recycled with the delay time t_r. Furthermore, let $f_{in}(L,t)$ be the mass flow density function of the particles fed into the system, and $\psi(L,t)$ be the selection function describing the classifying device. All of these flows are presented in Fig.A, illustrating schematically the structure of the whole continuous grinding system with classification and recirculation.

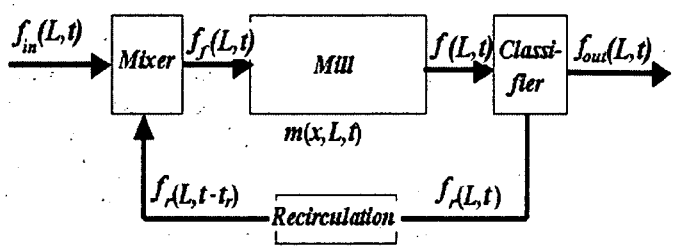


Fig.A. Continuous grinding system with classification and recirculation

Using the above notation, the mathematical model of the grinding mill can be written as a partial integro-differential equation:

$$\frac{\partial m(x,L,t)}{\partial t} = \frac{\partial}{\partial x}\left(D(x,t)\frac{\partial m(x,L,t)}{\partial x}\right) - \frac{\partial v(x,t)m(x,L,t)}{\partial x} - \tag{1}$$
$$-S(L,t)m(x,L,t) + \int_{L}^{L_{max}} S(l,t)b(L,l)m(x,l,t)dl$$

subject to the following initial and boundary conditions:

$$m(x,L,0) = m_0(x,L) \tag{2}$$
$$f_f(L,t) = v(x,t)m(x,L,t) - D(x,t)\frac{\partial m(x,L,t)}{\partial x}, \quad if\ x=0 \tag{3}$$
$$D(x,t)\frac{\partial m(x,L,t)}{\partial x} = 0, \quad if\ x=X \tag{4}$$

The left-hand side of Eq.(1) describes the rate of accumulation, the first term on the right-hand side represents the axial dispersion, while the second term corresponds to the convective flow of particles in the axial direction of the mill. The third and fourth terms of the right-hand side of Eq.(1) describe the rates of changes of particles due to the grinding process. The selection function $S(L,t)$ represents the rate of breakage of particles of size L. By means of function $b(l,L)$, called breakage density function, $b(l,L)\mathrm{d}L$ expresses the mass fraction of the product of size $(L, L+\mathrm{d}L)$ when particles of size l are broken. Based on that, $B(l;\lambda)$ denotes the breakage distribution function which expresses the mass fraction of particles of size l broken into the size interval $[L_{min}, \lambda)$, where L_{min} is the grindability limit of the mill. As a consequence, $B(l,\lambda) = \int_{L_{min}}^{\lambda} b(l,L)dL$.

The initial condition (2) expresses the fact that the mill is assumed to be filled with solids characterised with mass density function m_0 at the beginning of the process. Boundary conditions (3) and (4) reflect the continuity of particle fluxes at the inlet and that of the mass density function at the outlet of the mill, respectively. In boundary condition (3), the left-hand size term represents the flow from the transfer pipe into the mill, which is equal to the flow density of particles, consisted of convective and dispersive parts described by the right-hand side of Eq.(3). This boundary condition expresses the assumption that there is no back-mixing from the mill into the transfer pipe. At the exit boundary X, the particles are assumed to flow from a mixed region to a region where there is no mixing at all, so that the composition suffers no change here, and the boundary condition at the outlet reduces to the form of Eq.(4).

A continuous grinding system with classification and recirculation can be operated in various ways. Here we consider two models. In both cases, the material to be ground is fed into the mill through a mixer in which the mass flow $f_{in}(L,t)$ of

the fresh raw material, as well as the mass flow $f_r(L, t - t_r)$ of large size fractions recycled from the classifier, are mixed. In the first case, however, called model number I, the total mass flow of the fresh raw material is constant and does not depend on the amount of the recycled particulate product. Since the mass flow of the recycled material depends on the operating parameters of the mill and classifier, this mode, in general, leads to time dependent loading of the mill. This type of grinding system with classified product removal can be described by the following equations:

$$f(L,t) = v(X,t)m(X,L,t) \quad (5)$$

$$f_r(L,t) = \psi(L,t)v(X,t)m(X,L,t) \quad (6)$$

$$f_{out}(L,t) = [1-\psi(L,t)]\,v(X,t)m(X,L,t) \quad (7)$$

$$f_{in}(L,t) + f_r(L,t-t_r) = f_f(L,t) \quad (8)$$

Eq.(5) describes the mass flow density at the outlet of the mill. Eq.(6) expresses, by means of the selection function $\psi(L,t)$ characterising in principle the operation of the classifying device, that fraction of the particulate product that consists of particles having sizes larger than required and, as a consequence, are returned into the mill again. The quantity of returned particles therefore is determined by the selection function. Eq.(7) describes that part of the particulate product that is fine enough to leave the grinding process. Finally, Eq.(8) describes the mixture of particles resulted in mixing of the fresh raw material fed into the system and of the recycled mass flow by the mixing device. Also, Eq.(8) expresses that there is a time delay t_r in the recycled stream caused by the classifier device and the transport of particles through the recirculation line.

The second operation mode of continuous grinding systems with selective recirculation, considered here and called model number II, is as follows. The total mass flow of the material fed freshly into the grinding system is controlled in time according to the actual mass flow of the recycled particles, that may be variable in time depending on the operating conditions of the mill and the classifying device, in order to have constant loading of the mill itself. This type of operation of the continuous grinding system with selective recirculation is also described by Eq.(1) subject to the initial and boundary conditions (2)-(8), but it satisfy also the following constraint:

$$\int_{L_{\min}}^{L_{\max}} f_f(l,t)dl = constant. \quad (9)$$

where L_{max} denotes the maximal size of the particles to be ground by the mill.

The main difference between the two models is that model number I describes such a case when the mean residence time of the particles in the mill is varied depending on the actual size distribution of the fresh raw material fed into the system and on the operation conditions of the mill and classifying device. Conversely, in the second case the mean residence time of the material to be ground in the mill is constant by applying some appropriate control of the flow of raw material fed into the grinding system.

3 Discrete mathematical model of continuous grinding with classification and recirculation

In order to develop the computer models suitable for simulation of continuous grinding process with classification and recirculation, we had to discretise the partial integro-differential equation (1), describing the transport and breakage of the material to be ground in the mill itself, subject to the initial and boundary conditions (2)-(8). The discretization method used here is based on that developed and presented by Mihálykó *et al.* [13]. This type of discrete models has proved very useful in modelling and computer simulation of both batch and continuous grinding processes without classification.

Let us introduce the following notation. Let T denote the final time of a grinding process. We subdivide the interval $[0,T]$ into N equal subintervals, let τ denote the length of a subinterval, and let $t_n = n \cdot \tau$. We subdivide also the interval $[L_{min}, L_{max}]$ representing the extent of sizes of the whole particle population, into I equal subintervals, and let l_i denote the ith size fraction of those. The interval $[0,X]$ represents the length of the mill, and we subdivide it into J equal subintervals. Let h denote the length of a subinterval, and let $x_j = j \cdot h$. After the discretization process, we consider the mill as consists of imaginary columns. Let $V(x_j, l_i, t_n)$ denote the quantity of the particles belong to the ith particle size fraction in the jth column of the mill at t_n moment of time t. Further, let V_F denote the velocity of the material to be ground forward, while V_B the velocity of that backward in axial direction, respectively, so that $(V_F - V_B)$ is the velocity of the convective flow in the axial direction in the mill. Let a_i denote the quantity of the freshly fed particles of size l_i, r_i the rate of the returned part of the mass for for particles of size l_i leaving the mill, where $0 \leq r_i \leq 1$, and d the discretised time delay in the recirculation line.

As concerns the kinetics of breakage, usually the discretised versions of functions S and B, defined in Eq.(1) for the continuous case, are used for that purpose. Namely, here we chosen $S(l_k) = K_S \cdot (l_k)^n$ and $B(l_k, l_i) = (l_i/l_k)^m$, where $S(l_k)$ denotes the breakage selection function, representing the specific rate of breakage of particles of size l_k with parameters n and K_S, and $B(l_k, l_i)$ with parameter m denotes the breakage distribution function interpreted as fraction of breakage product from size interval l_k which falls into size interval l_i.

The discrete model number I of the continuous grinding system with classification and partial recirculation of product is as follows.

The quantities of particles in the first column of the mill at the moment of time t_{n+1} are expressed as

$$\begin{aligned} V(x_1, l_i, t_{n+1}) \;=\; & (1 - V_F) \cdot (1 - S(l_i)) \cdot V(x_1, l_i, t_n) + V_B \cdot V(x_2, l_i, t_n) \\ & + (1 - V_F) \cdot \sum_{k=i}^{I} p_{k,i} \cdot V(x_1, l_k, t_n) + a_i \qquad (10) \\ & + r_i \cdot V(x_J, l_i, t_{n-d}) \qquad\qquad i = 1, 2, \ldots, I \end{aligned}$$

where the first term on the right-hand side, $(1-V_F)\cdot(1-S(l_i))\cdot V(x_1,l_i,t_n)$, gives the amount of those particles from the ith size subinterval at the moment of time t_n which have not moved forward and have not broken. The second term, $V_B\cdot V(x_2,l_i,t_n)$ represents the quantity of those particles from the ith size subinterval at the moment of time t_n which have moved backward from the second column. The third term, $(1-V_F)\cdot\sum_{k=i}^{I}p_{k,i}\cdot V(x_1,l_k,t_n)$, expresses the quantity of those particles from the ith size subinterval at the same moment of time which have not moved forward from the first column to the second one, but have broken from some size subinterval to the ith subinterval of particle size. The term a_i is the quantity of the freshly fed particles belonging to the ith subinterval. Finally, the last term, $r_i\cdot V(x_J,l_i,t_{n-d})$, gives the quantity of the particles fed again into the mill because of their recycling.

The quantities of particles in some inner column of the mill can be described at the moment of time t_{n+1} as

$$\begin{aligned} V(x_j,l_i,t_{n+1}) &= (1-V_F-V_B)\cdot(1-S(l_i))\cdot V(x_j,l_i,t_n) \\ &\quad + V_F\cdot V(x_{j-1},l_i,t_n)+V_B\cdot V(x_{j+1},l_i,t_n) \\ &\quad +(1-V_F-V_B)\cdot\sum_{k=i}^{I}p_{k,i}\cdot V(x_j,l_k,t_n) \\ &\quad\qquad j=2,\ldots,J-1,\ i=1,2,\ldots,I \end{aligned} \tag{11}$$

The first term on the right-hand side of Eq.(11), $(1-V_F-V_B)\cdot(1-S(l_i))\cdot V(x_j,l_i,t_n)$, expresses the quantity of those particles from the ith subinterval at the moment of time t_n which have moved neither forward nor backward and have not broken. The second term, $V_F\cdot V(x_{j-1},l_i,t_n)$, represents the quantity of particles from the ith size subinterval at the moment of time t_n which have moved forward from the previous column. The third term, $V_B\cdot V(x_{j+1},l_i,t_n)$, gives the quantity of particles from the ith size subinterval at the moment of time t_n which have moved backward from the next column. Finally, the last term, $(1-V_F-V_B)\cdot\sum_{k=i}^{I}p_{k,i}\cdot V(x_j,l_k,t_n)$, represents the quantity of those particles which have moved neither forward nor backward from the jth column, but have broken from some size subinterval to the ith one larger from that.

At the last, the discretised process in the last column of the mill at the moment of time t_{n+1} can be given as

$$\begin{aligned} V(x_J,l_i,t_{n+1}) &= (1-V_F)\cdot(1-S(l_i))\cdot V(x_J,l_i,t_n) \\ &\quad + V_F\cdot V(x_{J-1},l_i,t_n) \\ &\quad + (1-V_F)\cdot\sum_{k=i}^{I}p_{k,i}\cdot V(x_J,l_k,t_n) \qquad i=1,2,\ldots,I \end{aligned} \tag{12}$$

The first term on the right-hand side of Eq.(12) expresses the quantity of those particles which have not moved forward from the last column, and they have not broken, i.e. there is no back mixing from the last column. The second term represents the amount of those particles which have moved forward from the last column, i.e. these particles have left the mill at the outlet at the moment of time t_n. The third term gives the quantity of those particles, which have not moved forward from the last column, but have broken from some particle size subinterval to the ith one. In Eqs (10)-(12) the meaning of symbols $p_{k,i}$ is, in principle,

$$p_{k,i} = \tau \cdot S(l_k) \cdot [B(l_k, l_i) - B(l_k, l_{i-1})] \text{ for all } k \text{ and } i.$$

The discrete model number II, considered in this paper, differs from that given by Eqs (10)-(12) only in the equation describing the processes occurring in the first column of the discretised version of the mill. Namely, the quantities of particles in the first column of the mill at the moment of time t_{n+1} are expressed as

$$\begin{aligned} V(x_1, l_i, t_{n+1}) &= (1 - V_F) \cdot (1 - S(l_i)) \cdot V(x_1, l_i, t_n) + V_B \cdot V(x_2, l_i, t_n) \\ &\quad + (1 - V_F) \cdot \sum_{k=i}^{I} p_{k,i} \cdot V(x_1, l_k, t_n) + a_i \qquad (10/a) \\ &\quad + r_i \cdot V(x_J, l_i, t_{n-d}) \qquad i = 1, 2, \ldots, I \end{aligned}$$

where now $a_i = a_i(t_n)$ $i = 1, 2, \ldots, I$ and

$$\sum_{i=1}^{I} (a_i(t_n) + r_i \cdot V(x_J, l_i, t_{n-d})) \text{ is constant.} \qquad (13)$$

Eq.(13) is the discretised version of Eq.(9). Since the second term on right hand side of Eq.(11), $V_F \cdot V(x_{j-1}, l_i, t_n)$, represents the influence of a column to the next one, the changes in the first column move smoothly to the other ones of the mill.

4 Simulation results and discussion

Based on the discretised equations (10)-(13), two computer programs, written in the language C, were developed for numerical experimentation. The size distributions of both the fed material and the initial loading material were chosen to be monodisperse in the simulation runs. The sizes of particles fed into the grinding system were chosen larger than $L_{max}/2$. At the same time, classification of the product was assumed to be total and sharp at particle size $L_{max}/2$, i.e. the selection function was chosen the Heaviside function of the form $\psi(L, t) = 1 \cdot (L - L_{max}/2)$ so that all particles larger than $L_{max}/2$ were returned from the classifier to the mixer.

The cumulative size distribution of the material being in the mill at coordinate x_j of the mill and that of the product leaving the mill were computed for all

moments of time t_n ($n = 0, 1, 2, \ldots N$) as the sum

$$M_j(L, t_n) = \sum_{l_i \leq L} V(x_j, l_i, t_n) \quad (j = 1, 2, \ldots, J-1), \quad \text{and}$$

$$M_J(L, t_n) = \sum_{l_i \leq L} V(x_J, l_i, t_n),$$

respectively. The oversize distribution functions were obtained by means of the relations $R_j(L, t_n) = 1 - M_j(L, t_n)$ ($j = 1, 2, \ldots J$-1) and $R_J(L, t_n) = 1 - M_J(L, t_n)$.

Eq.(1) describing the continuous grinding process may become independent of the time when $t \to \infty$. In this case, the stationary state is achieved when the dependence on time becomes negligible. Then, the size distribution and the total mass of the material leaving the mill also become independent on time. There may exist, however, such operation conditions of the grinding system, mostly due to the size dependent removal of the product, that the system does not achieve stationary state. In this case, the amount of the material to be ground in the mill increasing monotonically what leads to overloading of the mill after elapsing some time. Overloading is a heavy breakdown, and the mill must be stopped.

The simulation program was used to examine how the classification and partial recirculation, as well as the time delay in the recirculation line affect the stationary state and the oversize distribution of the product leaving the mill. The effects of changes of parameters both of the mill and the material to be ground were also examined. Let us see a few examples.

The effects of the classification and recirculation, and of the variation of the delay parameter d on the duration of transients and on some characteristic parameters of the size distribution of the particulate product is presented in Tables 1 and 2 for two different values of parameter K_s of the breakage selection function.

It is well seen from these tables that the duration of time required for reaching the stationary state is increased considerably with increasing the time delay, whilst the average size and dispersion of the size distribution are reduced. Table 1 contains simulation results for material the size reduction of which is easier, K_s=10^{-7}, than that of the material with parameter $K_s = 0.5 \cdot 10^{-7}$ the results of which are shown in Table 2. Comparing data in Tables 1 and 2 allows concluding that the extent of reduction of both the average size and the dispersion is smaller in the case of easy-to-grind material than in the opposite case. The extent of time delay influences the time required for reaching the stationary state, the average size and dispersion of the size distribution of the product significantly. The results obtained for cases d=3 and d=10 are very similar.

Using the same process and kinetic parameters as before, we obtained results shown in Tables 3 and 4 for the discrete model number II. It is seen that the effects of the recycling are not as much significant as in the case of the discrete model number I. In principle, in the case of the discrete model number II, the duration of time of transients is increased only to negligible extent, while only small changes can be observed in the average size and the dispersion of the size distribution of the material leaving the mill.

Table 1: Dependence of the duration of time required for reaching the stationary state, and of the characteristic parameters of the size distribution of the material to be ground at the outlet of the mill at the stationary state. Discrete model number I. Parameters: I=20, J=30, m=3, n=2, V_F =0.5, V_B=0.1, L_{max}=2000.

Recirculation, Delay	Ks	Time required for reaching the stationary state	The average size of the material leaving the mill	The dispersion of the material leaving the mill
No	10^{-7}	86	644.91	270.22
Yes, d=3	10^{-7}	145	635.15	267.45
Yes, d=5	10^{-7}	145	635.23	267.52
Yes, d=10	10^{-7}	145	635.38	267.68

Table 2: Dependence of the duration of time required for reaching the stationary state, and of the characteristic parameters of the size distribution of the material to be ground at the outlet of the mill at the stationary state. Discrete model number I. Parameters: I=20, J=30, m=3, n=2, V_F =0.5, V_B=0.1, L_{max}=2000.

Recirculation, Delay	Ks	Time required for reaching the stationary state	The average size of the material leaving the mill	The dispersion of the material leaving the mill
No	$0.5 \cdot 10^{-7}$	82	855.15	334.47
Yes, d=3	$0.5 \cdot 10^{-7}$	155	830.54	321.10
Yes, d=5	$0.5 \cdot 10^{-7}$	158	830.61	321.15
Yes, d=10	$0.5 \cdot 10^{-7}$	166	830.68	321.25

Table 3: Dependence of the duration of time required for reaching the stationary state, and of the characteristic parameters of the size distribution of the material to be ground at the outlet of the mill at the stationary state. Discrete model number II. Parameters: I=20, J=30, m=3, n=2, V_F =0.5, V_B=0.1, L_{max}=2000.

Recirculation, Delay	Ks	Time required for reaching the stationary state	The average size of the material leaving the mill	The dispersion of the material leaving the mill
No	10^{-7}	86	644.91	270.22
Yes, d=3	10^{-7}	87	639.54	265.43
Yes, d=5	10^{-7}	87	639.61	265.46
Yes, d=10	10^{-7}	88	639.38	265.59

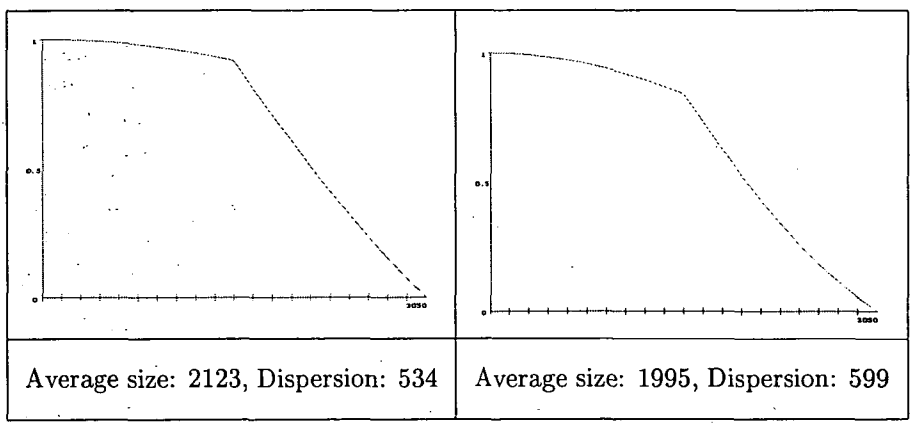
As a second example, we examine the oversize distribution of the leaving material as a function of time by using the discrete model number I. We compare the oversize distribution functions of the leaving material obtained by simulating the

Table 4: Dependence of the duration of time required for reaching the stationary state, and of the characteristic parameters of the size distribution of the material to be ground at the outlet of the mill at the stationary state. Discrete model number II. Parameters: I=20, J=30, m=3, n=2, V_F =0.5, V_B=0.1, L_{max}=2000.

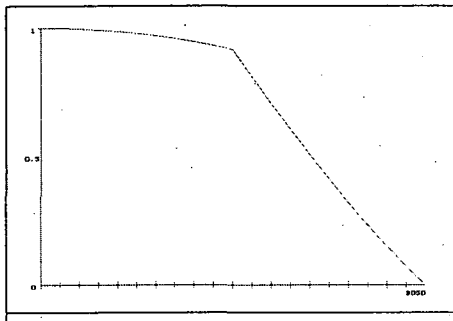
Recirculation, Delay	Ks	Time required for reaching the stationary state	The average size of the material leaving the mill	The dispersion of the material leaving the mill
No	$0.5 \cdot 10^{-7}$	82	855.15	334.47
Yes, d=3	$0.5 \cdot 10^{-7}$	86	840.42	323.00
Yes, d=5	$0.5 \cdot 10^{-7}$	86	840.55	323.09
Yes, d=10	$0.5 \cdot 10^{-7}$	87	840.18	323.22

behaviour of the grinding system with and without classification and recirculation. In these simulation runs we used process and kinetic parameters given in Figs 1-4.

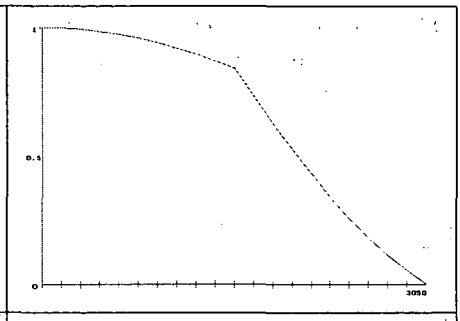
When grinding occurs without classification and partial recirculation of the product, the stationary state is reached at the t=76^{th} unit of the simulation time. In the case of applying classification and recirculation, however, the stationary state was reached at the 1153th unit of simulation time. The oversize distributions of the material to be ground at the outlet of the mill are shown in Figs 1-4 at t=10 and t=20 units of time. Figs 1-2 refer to grinding without recirculation. Comparing Figs 1 and 3, as well as Figs 2 and 4, it is seen that after a few units of time from the beginning of the process the oversize distribution functions are very similar to each other in both cases with or without recirculation. The effects of time delay and recirculation are hardly visible.



Figs 1-2. The oversize distributions of the particulate product at the outlet of the mill after t=10, t=20 units of time, respectively. Continuous grinding without recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F=0.5, V_B=0.1, L_{max}=3050, K_s= $9 \cdot 10^{-9}$.



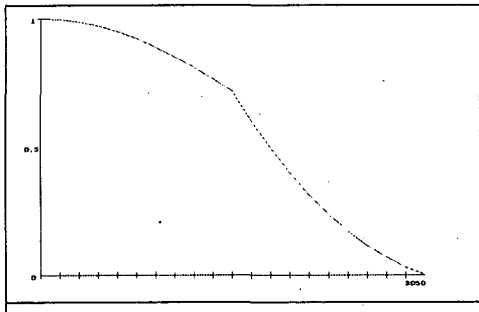
Average size: 2123, Dispersion: 545



Average size: 1995, Dispersion: 599

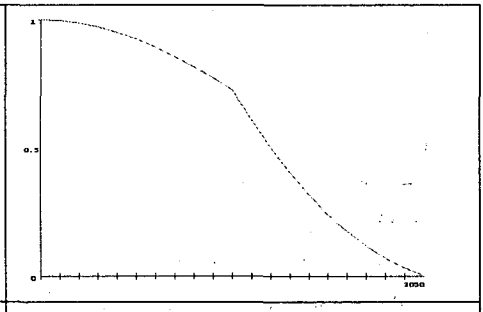
Figs 3-4. The oversize distributions of the particulate product at the outlet of the mill after t=10, t=20 units of time, respectively. Continuous grinding without recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F=0.5, V_B=0.1, L_{max}=3050, K_s= $9 \cdot 10^{-9}$, d=4.

Let us now see the oversize distribution functions at the 40^{th} unit of simulation time. Fig.5 refers to grinding without classification and recirculation, while Fig.6 refers to grinding with that. Here, we already see some differences between the two types of grinding mode. When the mill operates with recirculation the amount of particles belonging to the large particle size intervals is larger than in the opposite case. The oversize distribution function, shown in Fig.5, is similar to the oversize distribution function in stationary state, which is shown in Fig.10.



Average size: 1790, Dispersion: 644

Fig.5. The oversize distribution function shown at the outlet of the mill after t=40 units of time. Continuous grinding without recirculation.

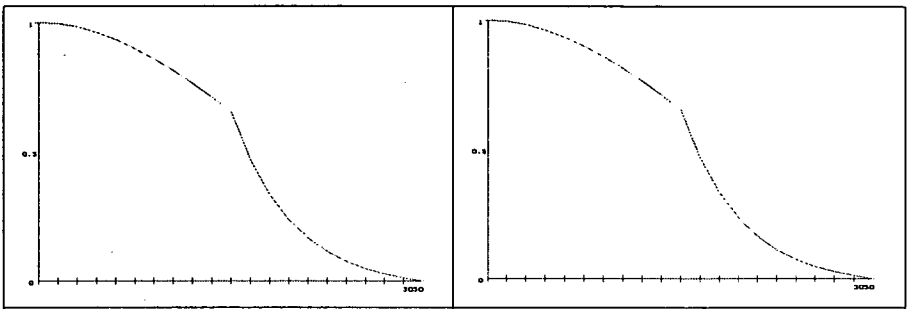


Average size: 1800, Dispersion: 642

Fig.6. The oversize distribution function shown at the outlet of the mill after t=40 units of time. Continuous grinding with recirculation.

In this example, the stationary state is reached after rather long time when grinding is carried out with recirculation. The composition is changed very slowly.

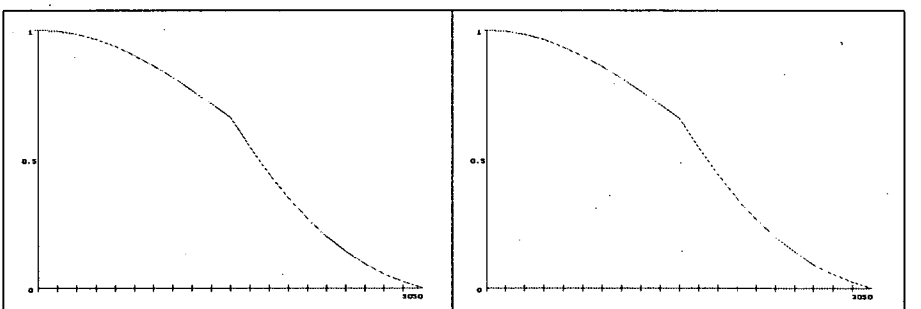
The oversize distribution functions are almost the same at t=600th, at t=900th and in the stationary state, as these are shown in Figs 7-8 and in Fig.12.



Average size: 1604, Dispersion: 582	Average size: 1603, Dispersion: 582

Figs 7-8. The oversize distribution functions shown at the outlet of the mill after t=600 and t=900 units of time, respectively. Continuous grinding with recirculation.

Close to the stationary state, the time evolution of the oversize distribution functions becomes very slow as it is shown in Figs 9-10 and in Figs 11-12.

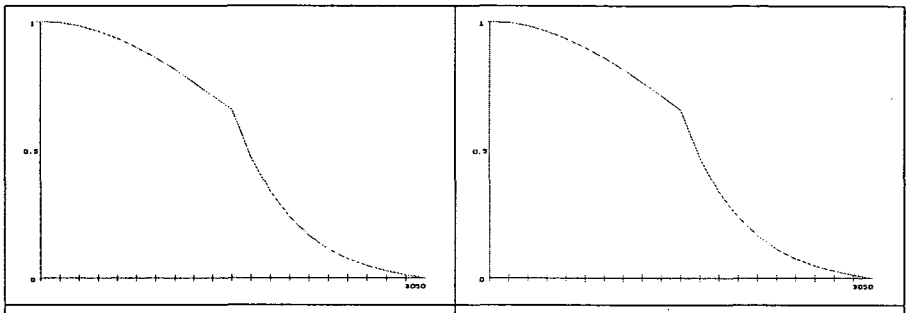


Average size: 1704, Dispersion: 654	Average size: 1700, Dispersion: 655

Figs 9-10. The oversize distribution functions shown at the outlet of the mill after t=66 and t=76 units of time, respectively. Continuous grinding without recirculation.

Figs 10 and 12 show that the stationary states are different in this case. When grinding was performed with classification and recirculation, the average size of the product was reduced with 5.7%. The time required for reaching the stationary state increased significantly due to the recirculation.

Let us now see what transient and stationary processes can be observed inside the mill. Since, by making the discretization of the partial integro-differential

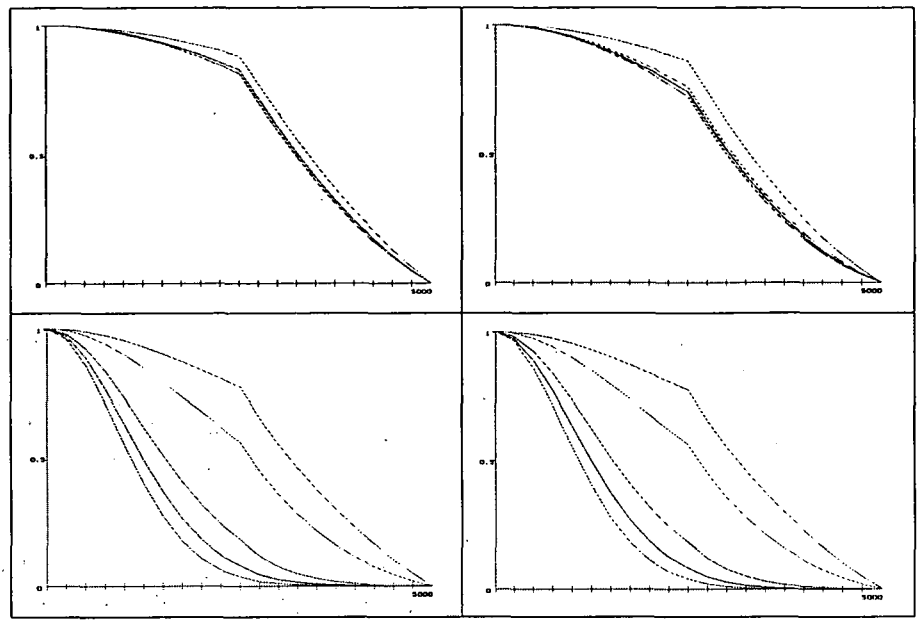


Average size: 1603, Dispersion: 581	Average size: 1603, Dispersion: 581

Figs 11-12. The oversize distributions shown at the outlet of the mill after t=1143 and t=1153 units of time, respectively. Continuous grinding with recirculation.

equation (1), we subdivided the length of the mill into 20 equal subintervals, subsequently called those columns, processes at different places inside the mill can be monitored as processes in the columns. The oversize distribution functions of the first, third, tenth, fifteenth and twentieth columns of the mill are shown in Figs 13-16, respectively, at t=5^{th}, t=8^{th}, t=150^{th}, t=287^{th} units of simulation time by using the discrete model number I. It is seen that the stationary state is reached at t=287^{th} units of time. Figs 9 and 10 show that after a few units of time the oversize distribution functions in the columns in question are very similar to each other since the initial loading of the mill and the size distribution of the fed material were chosen to be monodisperse. In stationary states, the size distribution of the material close to the inlet of the mill is quite different from that which is observed inside of the mill and near the outlet. The effects of the feed are seen only close to the inlet. The size distributions of the columns change only hardly approaching the stationary state. This is shown in Figs 15 and 16. In this case stationary state is reached very slowly.

The model number I, as it was defined before, describes such processes in the grinding system in which the total mass flow inside the mill, due to the dependence of the amount of the recycled material on the classifying device, may be changed. As a consequence, we can observe changes of the total mass flow in any column of the mill. The total mass flow of the particles in the first, tenth and last columns of the mill are shown in Fig.17 as a function of time. Due to the action of classification and recirculation, the total mass flow begins to increase in the first column of the mill after d=4 units of time Naturally in this case the time delay was 4 discrete time units. It is seen in Fig.17 that as the impulse of the total mass flow is shifted towards the outlet of the mill it is dispersed and the peak becomes smaller and smaller, i.e. the mixing process is smoothing the impulse. After reaching the new stationary state, the mass flow becomes equal and constant in time in each column of the mill, but this occurs at higher loading value.



Figs 13-16. The oversize distribution functions shown in the first, third, tenth, fifteenth and twentieth columns of the mill at t=5^{th}, t=8^{th}, t=150^{th} and t=287^{th} units of simulation time. Continuous grinding with recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F =0.3, V_B=0.1, L_{max}=5000, K_s=10^{-8}, d=4.

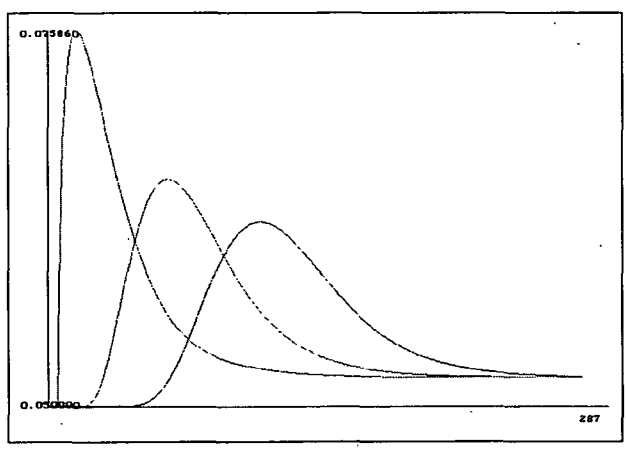


Fig.17. Variation of the total mass flow as a function of time in the first, tenth and last column of the mill, respectively. Continuous grinding with recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F =0.3, V_B=0.1, L_{max}=5000, K_s=10^{-8}, d=4.

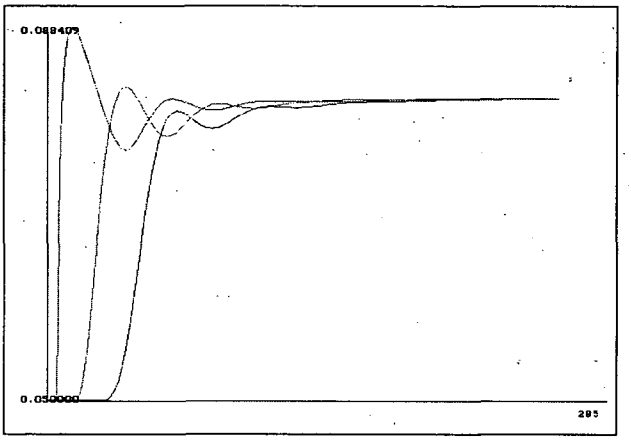


Fig.18. Variation of the total mass flow as a function of time in the first, tenth and last column of the mill, respectively. Continuous grinding with recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F =0.3, V_B=0.1, L_{max}=4400, K_s=10^{-8}, d=4.

In some case of simulation runs, damped oscillations of the total mass flow were detected as it is shown in Figs 18 and 19. In Fig.18, for instance, oscillations of the total mass flow in the first, tenth and last column of the mill are presented. In this case, in principle, oscillations become damped entirely after three decreasing characteristic peaks, and the mill reaches a stable stationary state at t=285^{th} units of simulation time.

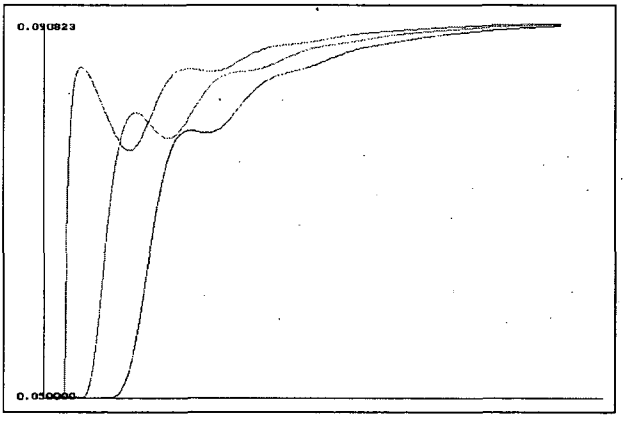


Fig.19. Variation of the total mass flow as a function of time in the first, tenth and last column of the mill, respectively. Continuous grinding with recirculation. Discrete model number I. Parameters: I=20, J=20, m=2, n=2, V_F =0.3, V_B=0.1, L_{max}=4000, K_s=10^{-8}, d=4.

Another example is shown in Fig.19, which is quiet different from the processes seen in Fig.18. In this case, the total mass flow exhibits oscillations with increasing peaks, although the amplitudes here are decreased, too, but after damping the oscillations the mass flow remains increasing monotonically.

In this case, the classifying device, because of the insufficient grinding efficiency of the mill, returns increasing amount of material to be ground again, thus, as a consequence, the load of the mill may increase above a given limit. It is an overloading phenomenon of the mill, and such grinding system is considered unstable.

5 Conclusions

Computer models and programs were elaborated for studying the stationary state processes of continuous grinding systems working with and without classification and partial recirculation of the product. The final form of the model was expressed as a set of recursive equations. The successive solution of the set of equations converges to the stationary state of the system.

The computer models developed are suitable for resolve a number of problems origin from the practice. For example, the problem of producing particulate product having prescribed average particle size with given constrains on the dispersion of the particle size distribution is a common one in process and mineral industry. By means of the newly developed programs it is possible to simulate the process in order to find the main properties and parameters of the grinding system which produces products satisfying the requirements of the end-users by efficient working of the grinding mill. These programs can also be used for estimating the kinetic parameters of the breakage processes, as well as for identifying the process parameters and conditions of the grinding devices and systems.

The efficiency of the grinding devices usually depends also on total mass flow and size distribution of the raw material fed into the system. The models and programs presented in the paper allow examining these effects, too. By simulating the transient processes caused by changes in the feed or recycled flows, the times required to reach the stationary states, as well as the conditions leading to overloading of the mill can be analysed and predicted, making possible to set the correct conditions of operation of the grinding systems in.

Acknowledgement. One of the authors (B.G.L.) would like to tank the Hungarian Research Foundation for supporting a part of this work under Grant *T034406.*

References

[1] G. Stoyan, Cs. Mihálykó, Zs. Ulbert, Convergence and nonnegativity of numerical methods for an integro-differential equation describing batch grinding, Comp. Math. Appl. 35 (1998) 69-81.

[2] Cs. Mihálykó, Z. László, É. O. Mihálykóné, On the qualitative properties of the analytical solution of an integro-differential equation, In: Volterra Equations and Applications (Ed. By C. Corduneanu and I. Sandberg), Gordon and Breach Science Publisher, Amsterdam, (1999) 351-356.

[3] H. Berthiaux, J. Dodds, A new estimation technique for the determination of breakage and selection parameters in batch grinding, Powder Tech. 94 (1997) 173-179.

[4] H. Berthiaux, J. Dodds, Modelling fine grinding in a fluidized bed opposed jet mill, Part I: Batch grinding kinetics, Powder Tech. 106 (1999) 78-87.

[5] V.K. Gupta, P.C. Kapur, A pseudo-similarity solution to the integro-differential equation of batch grinding, Powder Tech. 12 (1975) 175-178.

[6] L.G. Austin, R.R. Klimpel, The theory of grinding operations, Ind. and Eng. Chem. 56 (1964) 18-29.

[7] L.S. Gurevitch, Y. B. Kremer, A. Y. Fidlin, Batch grinding kinetics, Powder Tech. 69 (1992) 133-137.

[8] Y. Nakajima, T. Tanaka, Solution of batch grinding equation, Ind. Eng. Chem. Process Des. Develop. 12 (1973) 23-25.

[9] R. C. Everson, D. Eyre, Q. P. Campbell, Spline method for solving continuous batch grinding and similarity equations, Computer & Chemical Eng. vol.21, no.12, (1997) 1433-1440.

[10] H. Berthiaux, C. Chiron, J. A. Dodds, Modelling fine grinding in a fluidized bed opposed jet mill, Part II: Continuous grinding kinetics, Powder Tech. 106 (1999) 88-97.

[11] H. Berthiaux, D. Heitzmann, J. A. Dodds, Validation of a model of a stirred bead mill by comparing results obtained in batch and continuous mode grinding, International Journal of Mineral Processing, 44-45 (1996) 653-661.

[12] T. Blickle, Cs. Mihálykó, On a model of continuous grinding, Publ. of Dept. of Mathematics University of Veszprém, vol. 2/7 (1991).

[13] Cs. Mihálykó, T. Blickle, B.G. Lakatos, A simulation model for analysis and design of continuous grinding mills, Powder Tech. 97 (1998) 51-58.

Acta Cybernetica 15 (2002) 547–565.

Implemeting a Component-Based Tool for Interactive Synthesis of UML Statechart Diagrams

Johannes Koskinen*, Erkki Mäkinen†, and Tarja Systä*

Abstract

The Unified Modeling Language (UML) has an indisputable role in object-oriented software development. It provides several diagram types viewing a system from different perspectives. Currently available systems have relatively modest tool support for comparing, merging, synthesizing, and slicing UML diagrams based on their semantical relationships. Minimally Adequate Synthesizer (MAS) is a tool that synthesizes UML statechart diagrams from sequence diagrams in an interactive manner. It follows Angluin's framework of minimally adequate teacher to infer the desired statechart diagram with the help of membership and equivalence queries. MAS can also synthesize sequence diagrams into an edited or manually constructed statechart diagram. In this paper we discuss problems related to a practical implementation of MAS and its integration with two existing tools (Nokia TED and Rational Rose) supporting UML-based modeling. We also discuss information exchange techniques that could be used to allow the usage of other CASE tools supporting UML.

1 Introduction

The different diagram types provided by UML [23] have strong semantical dependencies. These dependencies allow, among other operations, slicing, synthesizing, and abstracting a UML diagram based on the information included in another diagram. A lot of tool support is available for constructing syntactically correct UML diagrams, but the present tools provide rather modest support for analyzing and using the semantical relationships of these diagrams.

In UML-based behavioral modeling, examples of object interactions are usually visualized as *sequence diagrams* or *collaboration diagrams*. The final specification of an object is modeled as a *statechart diagram*. A statechart diagram can be used as a protocol specification, showing the legal order in which the operations of an object may be invoked.

*Software Systems Laboratory, Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland, e-mail: jomppa@cs.tut.fi, tsysta@cs.tut.fi

†Department of Computer and Information Sciences, P.O. Box 607., FIN-33014 University of Tampere, Finland, email: em@cs.uta.fi

Automated support for constructing statechart diagrams from sequence diagrams provides considerable help for the designer. Automatic generation of state machines from scenario diagrams, such as UML interaction diagrams, has been studied extensively [7, 8, 12, 13, 25, 27].

MAS [10, 14, 15] is a tool that interactively synthesizes UML statecharts diagrams from sequence diagrams. Totally automatic synthesis algorithms, e.g., the one used in SCED [8], may result in a state machine that contains undesired generalizations. Because MAS consults the user during the synthesis process, the user can be confident that such "overgeneralizations" do not appear in the resulting statechart diagram. The user consultancy is organized as membership and equivalence queries posed by the algorithm. The user can help the synthesis process, for example, by marking certain (sub)paths in the statechart diagram appearing in a membership query as forbidden. This guarantees that the algorithm does not perform queries containing such a subpath more than once. We also consider various ways to support the user when she is providing a counterexample after rejecting a conjecture, i.e., after giving a negative answer to an equivalence query.

Tools like MAS, which support UML-based "model operations" [26], are desirable in all CASE tools. These techniques and tools can be integrated with CASE tools or they can be provided as separate components interoperable with CASE tools. One of our implementation platforms, the Nokia TED [28], is a multi-user software development environment that has been implemented at the Nokia Research Center. MAS interacts with TED through a COM interface. It imports the source sequence diagrams from and exports the resulting statechart diagram to the TED repository. The other implementation platform used is Rational Rose. In principle, MAS can be implemented for any tool supporting UML and providing a reasonable API for accessing the model repository. Moreover, commonly accepted exchange formats like XMI provide even more flexible integration of MAS with other CASE tools supporting UML. In addition to the diagram import and export mechanisms, the interactive nature of MAS brings additional challenges for the integration.

2 From Sequence Diagrams to Statechart Diagrams

In this section we briefly introduce the function of MAS (for further details, consult [10, 14, 15]). MAS tackles the problem of statechart diagram synthesis as a language inference task. The behavior of a selected participant described in a set of sequence diagrams is first mapped to strings belonging to the language to be inferred. MAS is then used to infer the language based on these strings. The resulting language is given as a finite state automaton. Finally, the automaton is transformed into a UML statechart diagram.

Before we discuss the actual synthesis algorithm, we briefly introduce a few aspects in UML sequence and statechart diagrams considered during the synthesis. The basic UML sequence diagrams to be considered in the rest of this paper consists

of participating *objects* and *messages* occurring between these objects. Objects are shown as vertical lines called *lifelines* and messages as horizontal arrows extending from a sender object to a receiver object. Let D be a sequence diagram describing a scenario with an instance I of class C. The *trace* originating from D with respect to I is obtained as follows. Consider the vertical line corresponding to I. Starting from the top, for two successive messages labeled e_i and e_j associated with I, where e_i is a sent message and e_j is a received message, add item (e_i, e_j) into the trace. If e_i or e_j is missing, then add $NULL$ instead. If the explicit deletion of the object is not shown at the end of the sequence diagram, let the right hand side of the last pair be $VOID$. The traces read in the above fashion are used as strings during the synthesis process.

In a UML statechart diagram, a transition from a state to another state can also be a so called *completion transition.* A completion transition without a guard is implicitly triggered by the completion of any internal activity in a state [23]. Therefore, in MAS, we do not allow a state to have both labeled and unlabeled transitions (the latter corresponding to unguarded completion transitions) as outgoing transitions. In MAS we do not allow a completion transition and a labeled transition to be the leaving transitions of the same state. Two leaving completion transitions, in turn, would result in a nondeterministic state.

2.1 The Algorithm

Being a minimally adequate teacher requires that the designer can answer two kind of simple questions:

1. she must decide whether a given behavior is possible in the system she is implementing (the membership queries)
2. she must accept or reject the output statechart diagram, and moreover, if she rejects, a counterexample from the the symmetric difference of the languages related to the output statechart diagram and the desired statechart diagram must be given (the equivalence queries).

In addition to definite *Yes* and *No* answers, MAS allows the user to answer *Maybe* or *Hardly* (i.e., weak *Yes* and *No* answers). The information obtained from these answers is considered less significant than that obtained from normal, definite answers. Furthermore, the user can postpone answering by saying *Later.* These inaccurate answers are discussed in greater detail in [10].

MAS maintains an *observation table* T containing the current information about members and non-members of the desired language. The rows of T are labelled by the elements of $(S \cup S \cdot A)$ where A is the input alphabet, and S is a prefix-closed set of strings in A^*. (Notice that the alphabet A of our inference algorithm consists of pairs (e_i, e_j) and that "$\cdot$" stands for the concatenation operator.) The columns of T are labelled by the elements of a suffix-closed set R. MAS generates the columns during the synthesis process. They contain possible continuations to strings in the rows and are used to decide whether the rows represent paths that yield to the same state. In other words, the columns are used to test if two states can be joined. The

entry for row s and column r, is 1, if $u = s \cdot r$ is in the desired language, otherwise the entry is 0. The bit string on the row labeled x in T is denoted by $row(x)$. An observation table is said to be *closed* if for each t in $S \cdot A$, there is an s in S such that $row(s) = row(t)$. An observation table is *consistent* if whenever s_1 and s_2 are in S such that $row(s_1) = row(s_2)$, for all a in A, $row(s_1 \cdot a) = row(s_2 \cdot a)$.

The original inference algorithm [2] starts with $S = R = \emptyset$ and first asks membership queries for λ (the empty string) and for all symbols a in A. T is updated by the answers of these queries. While T is not closed and consistent, new strings are added to S and R, and the corresponding table entries are found out by membership queries. A closed and consistent observation table defines a deterministic finite automaton in a natural way [2]. The algorithm forwards this automaton as a conjecture to the teacher. The algorithm halts if the teacher accepts the conjecture. Otherwise, the given counterexample updates T and the execution of the algorithm continues.

In our application, where the designer plays the role of the teacher, the execution of the algorithm begins so that the designer constructs a set of typical sequence diagrams describing the behavior of the system. All traces from these sequence diagrams and their prefixes are stored in S. No membership queries are needed since the traces themselves are in the unknown language but all the proper prefixes are not. Indeed, if a string ends with a symbol (e_i, e_j) with $e_j \neq VOID$, the membership query is not necessary since we know that the string in question cannot belong to the unknown language.

There are also other application specific features in the synthesis process that decrease the number of membership queries needed. Consider now a trace

$$e = (e_1, e_2)(e_3, e_4) \ldots (e_{i-2}, e_{i-1})(e_i, e_{i+1}) \ldots (e_{n-1}, e_n),$$

which is in the unknown language. Since e is in S, then so have its prefixes including $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})$. The left hand side e_i of (e_i, e_{i+1}) defines the action in the state reached by the subtrace $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})$. Hence, we do not have to make membership queries for strings $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})w$, where $w = (e_j, e_{j+1}) \ldots (e_{m-1}, e_m)$ and $e_j \neq e_i$.

The algorithm outputs a finite automaton. Actually, we need a statechart diagram which is obtained by fine tuning the output automaton as described in [15]. The output finite automaton is called the *underlying finite automaton* of the resulting statechart diagram.

2.2 Data Structures Allowing Backtracking

MAS allows the user to give inaccurate answers to membership queries. Thus, we need data structures that are able to manage the user's mind changes or accidentally given incorrect answers.

MAS maintains a trie containing the strings known to be in the desired language. (For the definition and basic properties of tries, see e.g., [16].) This trie is referred to as W. Initially, W contains the information related to the input set. New

information is inserted in W when the user gives a positive answer to a membership query, or when she gives a counterexample not belonging to the language accepted by the conjectured automaton. We need a trie structure in which we are able to efficiently backtrack, and then update the observation table if necessary, i.e., we use a structure that resembles so called persistent data structures (see [6]).

Suppose now that MAS is looking for the correct value for an entry in the observation table T. It first checks that the string (say w) in question ends with a symbol of the form $(e, VOID)$. If so, it accesses W and compares the existing links against w. There are three different possibilities:

1. the links can be traversed to a leaf, which means that the trie contains w; the correct entry in T is 1,
2. w is of the form $w = w_1(e_i, f)w_2$, where w_1 is the longest possible common prefix of w and any string (say y) in W, and y continues with a symbol (e_j, g) where $e_i \neq e_j$; now we know that w cannot belong to the desired language and the correct entry in T is 0, and
3. w is of the form $w = w_1(e_i, f)w_2$, where w_1 is the longest possible common prefix of w and any string y in W, and y continues with a symbol (e_j, g) where $e_i = e_j$ (and hence, $f \neq g$); MAS cannot conclude the correct entry in T, and a membership query is needed.

The algorithm tries to determine table entry values without consulting the user. We prepare ourselves to possible backtrack operations by maintaining pointers in order to reach the table entries whose value is determined by the algorithm. If a trie node used in determining table entries is later deleted, these entries can be easily found by following the pointers.

Inserting new elements to the trie is as straighforward as accessing. However, problems arise when we have to delete a string from the structure because of a found error or of a mind change of the user. The deletion itself is easy, but it is possible that we have updated the observation table based on the existence of a string, which now turns out to be erroneous. Hence, we have to check that the value of all observation table entries are determined from existing trie elements also after the deletion.

Consider now what happens when a string is deleted from the set of words known to be in the desired language. First, the corresponding element is deleted from W. The algorithm might have concluded an affirmative answer to a membership query based on the (now ceased) existence of the string in question. Now, this entry in the observation table must be updated to be 0. The possible need for reconsidering the value of a table entry can be concluded by checking the lists of coordinates along the path presenting the element to be deleted from the trie structure. Notice, however, that a change in the value of an observation table entry is not necessarily needed. The string corresponding to the observation table entry in question may contain other substrings, from which a negative answer can be concluded (or the user can confirm by answering a membership query that the entry should be kept unchanged).

2.3 Improving the Algorithm

Depending on the case tool MAS is integrated with, the performance of the synthesis process varies. According to our studies, in the case of TED, importing and exporting diagrams to and from the TED repository is the most time consuming part with small and moderate size examples (excluding the time spend with user interactions) [9]. With large examples, in turn, also the performance of the algorithm becomes an issue.

An obvious direction for improving the performance of MAS is to further decrease the amount of user consultancy. There are two different lines to follow. First, we can equip the user with methods to transfer her knowledge about the system to the algorithm. These methods are introduced in Section 3. Second, we can try to make use of the general improvements suggested to Angluin's original algorithm in the literature.

In what follows, we shortly discuss the suggestions by Rivest and Schapire [24] (see also a survey by Balcázar *et al.* [3]). The idea of Rivest and Schapire is to use a "characteristic member" of each class of strings in S with an equal row. This means that the observation table is always consistent. Clearly, the principle also decreases the size of the observation table, and as a consequence, the number of membership queries is decreased too, at least in the worst case. The crux of the improvement is the handling of counterexamples. Instead of inserting the counterexample and its prefixes to S, the method of Rivest and Schapire finds out a new member for R. This string is chosen so that it makes the observation table non-closed, and in order to retain closeness, a prefix of the counterexample is inserted in S. Although membership queries are needed to find the correct prefix of the counterexample, it can be shown that this method indeed decreases the number of membership queries in the worst case. However, it is still open whether this method actually decreases the number of membership queries in our application. Namely, it is essential how many membership queries the algorithm can answer without consulting the user. The queries induced by the method of Rivest and Schapire may be difficult for the algorithm.

If applied in its basic form, the method of Rivest and Schapire has the drawback that the new conjecture does not necessarily classify the previous counterexample correctly [3]. This feature is not acceptable since it would confuse the user by making the user interface illogical. However, this problem can be settled by not showing the new conjecture to the user and using the same counterexample as long as the counterexample is not correctly classified. This would also decrease the number of equivalence queries.

It is even known that membership queries are not necessary at all for a polynomial time inference algorithm for regular languages, provided that the teacher always gives (lexicographically) smallest counterexamples (see e.g., Birkendorf *et al.* [4]). However, this result does not help us, since it is unreasonable to expect the user to provide smallest counterexamples to the algorithm. Still the choice of the counterexamples does have its effect to the efficiency of MAS: short (positive) counterexamples are, of course, desirable.

There are also various ways to streamline the data structures. For example, the observation table is very sparse, i.e., a great majority of its entries are zeroes. This fact can be utilized by storing only the entries containing ones.

3 Interaction Between the User and MAS

In this section we discuss the information exchange and visualization techniques between MAS and the user. We introduce various methods for transfering additional information to the synthesis algorithm in order to further decrease the number of membership queries. For the membership queries to be informative and interesting enough, they need to vary from each other. Moreover, the amount of queries should be considerable small not to cause the user to lose her interest. Ideally, MAS should draw the user's attention to crucial and ill-defined parts of the current design but not bother her with trivial questions. In what follows we sketch techniques that allow the user to give "general guidelines" to MAS, thus decreasing the amount of queries. Especially, the proposed techniques aim at decreasing the amount of similar or closely related questions, answers to which depend on the same key question.

3.1 Visualizing the Membership Queries

Choosing an appropriate information visualization technique is important in interactive systems. A membership query needs to be shown to the user in a way that is easy to understand and answer. An intuitive way to visualize a membership query would be highlighting the corresponding path in a statechart diagram. However, since some of the membership queries are posed before a conjecture for a statechart diagram can be represented, this is not possible for all membership queries.

Currently, MAS poses the membership queries in a form of a simple sequence diagram with two participants: the object of interest and a participant (called *System*) that represents all other participants (inside or outside the system border) the object interacts with. A membership query often consists of subpaths that have already been accepted by the user. In such a case, the membership query can be translated to a question: "Can these subpaths occur in the presented order?". To make it easier for the user to recognize such components (subpaths) in the membership query, MAS uses a different color for each one of them. Figure 1 shows a sample membership query. It consists of two subpaths already accepted by the user. The equivalence queries, in turn, are visualized as statechart diagrams by the CASE tool (currently TED or Rose) itself.

In order to give the user a flexible way to express her mind changes concerning the status of a piece of inaccurate information, MAS provides a window in which all the inaccurate information is presented. The user can browse the questions and modify the answers simply by clicking the mouse button. If the user now gives an accurate answer, the question disappears from the window. This window is opened when the user gives an inaccurate answer (*Maybe* or *Hardly*) for the first time.

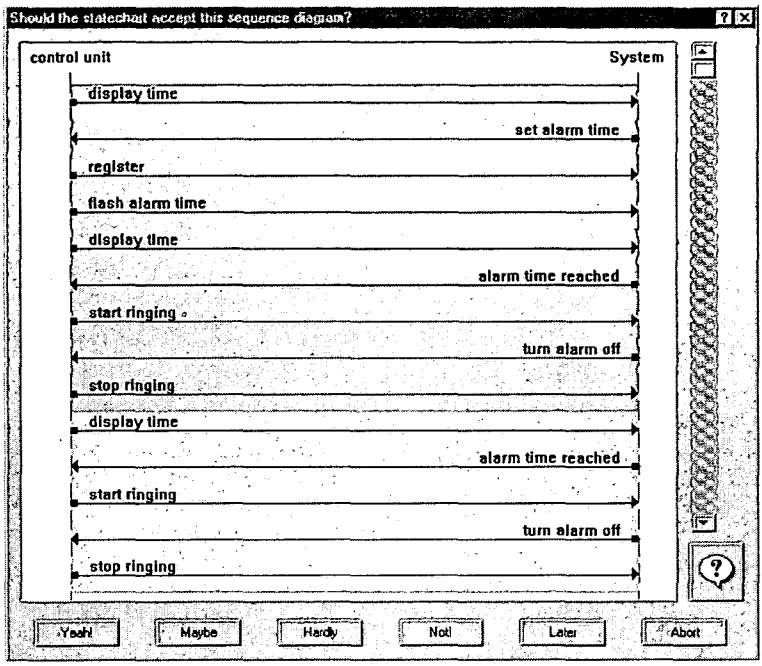


Figure 1: A sample membership query

3.2 Forbidden Substrings

It is obvious to the user of MAS that certain sequences of messages cannot take place, or equivalently, that certain substrings are not possible in the words belonging to the desired language. It is, however, quite unreasonable to expect that the user can list such invalid subpaths beforehand. A user-friendly way to transfer this information to the algorithm is to give the user a possibility to mark any subpath of a membership query as invalid. This guarantees that the algorithm does not make membership queries with the same invalid subpath more than once. Such a possibility increases the generality of the answers: instead of neglecting a single word from the unknown language, we can neglect a whole sublanguage of words containing the invalid pattern. For example, from the membership query in Figure 1 the user might want to select a block from the sixth message (alarm time reached) to the 12th message (start ringing) as a forbidden subpath, indicating that an alarm clock should not start ringing if the alarm is not set on (even though the alarm time is reached).

We need another trie (referred to as F), which contains the forbidden substrings. It is accessed if the correct answer cannot be concluded based on the information stored in W. In the nodes of F, there are lists of pointers to the observation table entries whose values are concluded from the trie element in question. Since

deletions should be possible also from F, it is maintained analogously to W, i.e., a deletion may cause changes in the observation table entry values. Checking whether a given string contains any of F's strings as its substring, is an instance of a string matching problem where several patterns are searched from a single text.

3.3 Editing the Statechart Diagram

Answering equivalence queries and providing a sufficient set of sequence diagrams as counterexamples can sometimes be quite tedious when defining the correct statechart diagram. The user should have a more direct method to change the conjecture. A typical object-oriented design tool allows the user to edit the statechart diagram by adding new states and transitions, by deleting existing ones, and by splitting and merging states.

So far, we have considered the construction of an automaton from a given consistent and closed observation table. When the user is allowed to edit the statechart diagram, we have to have a method for traversing also to the opposite direction from the statechart diagram (or from the corresponding finite automaton) to an observation table that defines the original finite automaton / statechart diagram.

Even a small editing operation in the statechart diagram may cause a major change in the observation table. Furthermore, the whole statechart diagram might have been constructed manually. Hence, we obey the policy to always build up the observation table from scratch. This is possible by using the algorithm *BuildUp* introduced in [14]. It is clear that the observation table can be filled up without consulting the user. Moreover, the observation table obtained is closed and consistent, and it defines the edited statechart diagram.

3.4 Providing Counterexamples

The task of providing counterexamples is the most difficult part of using MAS. Hence, the user interface should support the user to find proper counterexamples and to check their consistency with the other information available.

Suppose that MAS has output a statechart diagram with B as the underlying finite automaton and that the user does not accept the conjecture. The user is now expected to provide a counterexample. If she gives a positive counterexample w, i.e., a string not in the language $L(B)$ accepted by B, MAS should change the conjecture so that w is contained in $L(B)$. Otherwise, the user gives a negative counterexample (a string w in $L(B)$) and MAS should omit w from $L(B)$.

The normal way to give a positive counterexample is to present an extra sequence diagram. When the user gives her counterexample, the interface should confirm whether or not it is in $L(B)$, so that she can be sure that the counterexample is of the desired type. An instructive way of telling this is to animate the function of the conjectured statechart diagram with the input w. This ensures that the counterexample has the desired effect to the statechart diagram. In our approach, the conjectured statechart diagram is visualized by the CASE tool. This means that the API of the CASE tool in question should allow its extension with

such animation property. Considering our current implementation environments, the Rational Rose Extensibility Interface (REI) allows this while the TED API does not.

The task of giving a negative counterexample is often more natural to replace by editing the statechart diagram. For example, deleting a transition from the statechart diagram is equivalent with giving a set of negative counterexamples, which are now longer accepted by the statechart diagram when the transition is missing.

An easy method to define a very general type of negative counterexamples is to allow the user to select paths from the conjectured statechart diagram by clicking its states on the screen. Suppose the user clicks a pair of states s_1 and s_2 one after another. This can be interpreted so that all paths from state s_1 to state s_2 are forbidden. In other words, all the substrings of the form $(a, x)y(b, z)$, where a and b are the actions related to the states s_1 and s_2, respectively, x and z are any messages, and y is any sequence of pairs, are forbidden. Hence, by clicking states we can define even more general classes of strings as forbidden than by marking substrings in membership queries. Again, the API of the CASE tool should allow activation of the states.

4 Integrating MAS with CASE Tools

A variety of CASE tools supporting UML is available. These tools provide syntactic support for UML-based software development. Moreover, code generation and/or basic round-trip engineering facilities (typically limited to relations between a class diagram and source code) are supported by many of these tools. A more interesting and more challenging problem is to provide semantical support for applying operations among different UML diagrams. Selonen *et al.* [26] divide *model operations* into two groups: (1) *basic operations* that apply set theoretical operations (union, difference or intersection) for two diagrams of the same type and (2) *transformation operations* that take a UML diagram as an operand and produce a diagram of another type as its result. Both basic and transformation operations involve the semantics of the diagrams and need a case tool for providing the information content of the diagrams and for visualizing the resulting UML diagram. Thus, model operations could be implemented as separate components that provide import and export services for the tools (supporting UML) they are interoperating with.

For managing interoperability, the information exchange format should be agreed on. In what follows we discuss the advantages and disadvantages of different integration techniques from the point of view of MAS.

4.1 Integrating MAS with TED

The TED version of MAS is implemented as a stand-alone program, which connects to the TED's repository using a special TED COM server. This server is located in a local computer and it establishes a connection to a remote TED server. The

user needs to specify where the server and the repository are located and how the traces to be synthesized can be found.

Additionally, the user needs two different tools, one for constructing a sequence diagram and another for synthesizing the statechart diagram, and to switch between them during the synthesis process. This is not practical with interactive synthesizers like MAS. When MAS asks for a counterexample, the user needs to make a sequence diagram or to modify the existing statechart diagram. When she is ready, MAS has to know the exact location of the diagram (ID or path to it) before it can continue. There is no way the user can point an object in the editor and tell MAS to continue synthesizing using that trace as a counterexample.

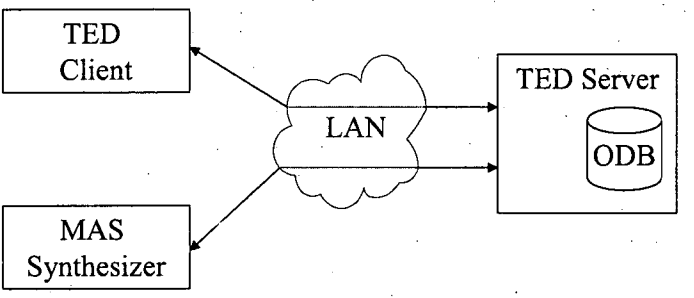


Figure 2: The TED implementation of MAS

4.2 Implementing a Software Component

Programming languages offer mechanisms to distribute and reuse software libraries, but these techiques have been vendor and language specific and communication between different libraries has been difficult. The object component model is a language and vendor independent mechanism to reuse existing software.

In Microsoft Windows environment we can choose between two object component technologies: The Common Object Request Broker Architecture (CORBA) [22] and The Component Object Model (COM) [17]. Information about the differences between these two technologies can be found in [5]. COM is designed for Windows platform. Almost all CASE tools offer some kind of COM interface for automating design. In addition, with COM Automation [18] we could use our synthesizer tool from scripts and macros.

COM is a platform independent, distributed, object-oriented system for creating binary software components that can interact. These components (objects) can be within a single process, in different processes, or even on remote machines [19]. Every component has an unique identifier (called Globally Unique Identifier, GUID) and information about available components is stored in the system registry.

The main idea of the software components is that only interfaces are provided for their users, their implementation is hidden. The COM components can be used

like normal C++ classes, but the code itself may be executed on a remote machine – the client application does not have to worry about the components' location. There are some guidelines to specify a COM component [20]:

- **A COM interface is not the same as a C++ class.** The pure virtual definition carries no implementation. Unlike C++ classes (interfaces), the COM interfaces cannot have any implementation.
- **A COM interface is not an object.** It is simply a related group of functions. It is also the binary standard through which clients and objects communicate.
- **COM interfaces are strongly typed.** Every interface has its own interface identifier (a GUID), which eliminates the possibility of duplication that could occur with any other naming scheme.
- **COM interfaces are immutable.** You cannot define a new version of an old interface and give it the same identifier. Thus, each interface is a separate contract, and systemwide objects need not know whether the version of the interface they are calling is the one they expect. The interface ID (IID) defines the interface contract explicitly and uniquely.

The COM components can communicate with the client software using events.

4.3 Integration Considerations

The most trivial and flexible way to manage interoperability is to change information through files written in a predefined format. For an optimal interoperability, a file format supported by several CASE tools should be chosen. A downside of this approach is inefficiency: additional reading/writing information from/to files is time consuming compared to the direct use of the information. Since the tool that provides the information need not to be the same as that visualizing the results, this approach allows, for instance, the source sequence diagrams to be constructed with multiple tools.

XML-based Metadata Interchange (XMI) [21] is an interchange format for UML supported by most of the case tools supporting UML. Since the Document Type Definition (DTD) grammar that defines the XMI language is based on UML metamodel (or more precisely, on MOF (Meta-Object Facility) specification [21]), it can only express what is in the UML metamodel, thus lacking support for defining presentation information (e.g., layouts). However, using the extension mechanism of XMI, the tool vendors can define how to add that information to the XMI files. Since there currently does not exist a global agreement on how this should be done, the UML CASE tools can only exchange model information in practise. From the point of view of MAS, this is not a crucial problem, since the imported statechart diagrams are created by MAS and thus, they do not contain any history information on the diagram layouts to be restored.

Integrating MAS more tightly with different CASE tools allows us to extend the possibilities to communicate with the synthesizer. The user can start the synthe-

sizer from the menu and all interactions can be managed using one tool (although in separate windows). While editing statecharts is a relatively easy way to express the counterexamples, the editor can support this task by previewing the conjecture and allowing the user to modify it before she continues the synthesis process.

Using repository via a server has a performance penalty as well. For example, generating a statechart to the repository takes a long time (about one second per state in our case). When the synthesizer and the editor both use the same internal data format, the repository becomes obsolete as temporary storage for synthesized conjectures. Only the final conjecture should be placed in the repository for future use.

Our TED implementation of MAS allows semi-automatic synthesis using startup parameters. The initial sequence diagrams can be given parameters without any interaction, but membership queries and counterexamples will still need user consultation. The startup parameters can also be given graphically using the visual scripting mechanism in TED [11].

To integrate MAS with CASE tools we need to build MAS as a software component. This component implements an interface offered by the CASE tool and either uses a specific interchange format (e.g., XMI) or a special interface for exchanging UML models between the component and the tool. Using an internal data format is a simpler and faster solution, but it limits us to use the single specific CASE tool. XMI is a universal format, but exporting and importing XMI files (even memory mapped files) might slow down the performance drastically with large data structures, especially when chaining the components. In some cases, using pre-saved XMI files allows us to speed up the synthesizing, but we have not made a speed comparison between these two techniques.

A MAS COM component should provide a high-level synthesis interface to start and control the synthesizer from CASE tools. The interface should have at least the following methods:

- `Synthesize(IDataInterface* in, IDataInterface* out)`
  A very high-level synthesis method to start a synthesizer. The input and output interfaces (*IDataInterface*) are used to get a sequence diagram and to put a conjectured statechart diagram back to the editor. This interface is similar to startup parameters with the exception of events. Using this interface method, the synthesizer can notify the editor in different synthesis phases.

- `InsertCounterExamples(DWORD count, IExampleInterface** examples)`
  Inserts a number of counterexamples (sequence diagrams or edited statechart diagrams) for the synthesizer. This method is used after the synthesizer has notified the editor to give a counterexample with an event.

- `InsertForbiddenStrings(DWORD count, BSTR* strs)`
  Tells MAS not to accept traces (strings) by default. This is useful when handling the forbidden strings considered in Section 3.2.

All input and output data is exchanged using interfaces. These interfaces provide required methods to access internal data structures. The data itself could be in XMI files, internal data format, or in server's repository. In addition to the methods, we need several events to tell the editor (or more generally, the client) when the synthesizer will need user action. This allows the client to set *callbacks* and to modify the synthesizer's behavior and user interface.

4.4 First Rays of the New MAS

We have already made the first version of MAS for Rational Rose (later *roseMAS*) using the techiques described in Section 4.2. The synthesizer is a COM component which is connected to the modeling software using *Rational Rose Extensibility Interface* (REI). REI is a COM automation interface for various plugins. It contains methods to manipulate Rose models (e.g., creating new elements) and to extend the user interface (like additional menus). Although roseMAS has been designed to be a Rose plugin, it can also be used with other tools because of its automation interface. On the other hand, roseMAS component cannot be run without additional command line utility.

When roseMAS has been installed using the setup program, it registers itself to the Windows registry, so that CASE tools can use it. The registry contains information on events that roseMAS is interested in, such as selecting a menu item from Rose. When an event occurs, Rose calls a method in roseMAS interface *EventHandler* (see Figure 3). The roseMAS installation package also includes a menu file, allowing MAS options to be changed directly from the CASE tool.

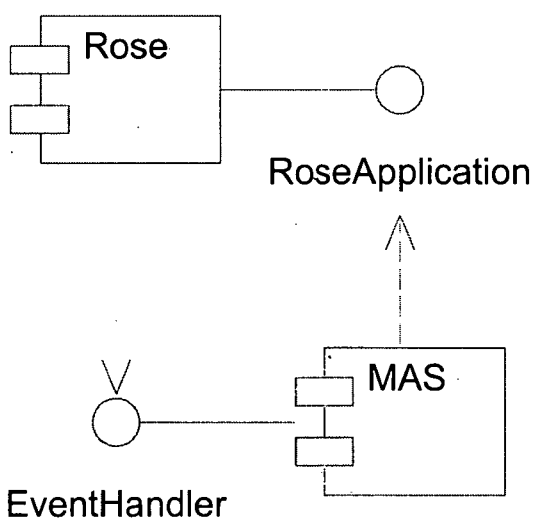


Figure 3: The component diagram describing the implementation of MAS for Rose

When the user selects the *SED→SCD* command from the pop-up menu, roseMAS looks up all selected and active items. One of them has to be an object. The other selected items are sequence diagrams, which contain the same object (or at least object with the same name). RoseMAS gets all the messages related to the object of interest from diagrams and adds them to the trace list. After that, the original MAS algorithm starts with these input traces.

All communication between roseMAS and Rose is managed via *RoseApplication* interface. Since the interface supports automation, we can use a C++ wrapper class to hide COM specific code and use the RoseApplication like a class library.

Comparing this Rose integration to the one with TED, the differences between these two are remarkable. Instead of giving startup parameters and typing location information to the dialogs, the user can select the sequence diagrams she would like to include in the synthesizing. Starting roseMAS is easy because of the direct menu support (see Figure 4). Furthermore, the conjecture is automatically created under the base class of the traced object. The user can give a counterexample, like starting the synthesis. This is accomplished by selecting sequence or statechart diagrams and pushing the *continue* button.

In addition to all this, the performance of the new roseMAS is much better than that of the old client-server system. On the other hand, Rose lacks a multi-user collaboration and database system. This means that we are back with the "one model per file" environment. Moreover, unlike TED, Rose tries to keep our model and diagrams consistent. Normally, this is what the user wants, but when the user synthesizes new diagrams, consistency checking makes some things a bit unconfortable. For example, the states of the generated statechart diagrams can no longer use the same name between diagrams, because they share the same state machine.

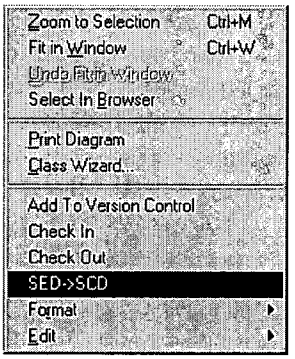


Figure 4: The MAS can be started from the menu

5 Future Work

Currently, we have integrated MAS with two different CASE tools. It would be useful to separate a CASE tool and the synthesizer with a tool-independent platform, so that we would need only one implementation of MAS for all the CASE tools supported by the platform. Such a platform (xUMLi, executable UML interface) has been introduced in [1]. When integrating MAS with xUMLi some problems might appear because of the interactive nature of MAS.

The main problems with MAS are in the user interface. We should equip the user with more efficient methods to transfer her knowledge to the algorithm, and the algorithm should have better ways to support the user in making various decisions. In addition to the topics discussed in the previous sections, at least the following things call for our attention.

We introduced uncertain answers, which allow the user to change her mind later. After giving an inaccurate answer the user might be interested in what part of the current membership query the uncertain portion is (i.e. if the user has given an uncertain answer to query A and/or query B, the dialog should show the uncertain part in query ABC). This could be indicated by using appropiate colors in the queries. In addition, the conjecture generated by MAS could distinguish uncertain and certain paths same way as in the situations mentioned above.

When synthesizing complex systems (e.g., a dialog with buttons and other control elements), it would be helpful, if all components were synthesized at the same time. The resulting conjecture would have multiple statecharts with hyperlinks between different synthesized components allowing the user to switch between generic (the statechart from the dialog) and more specific (the statechart from the button) view. Some kind of 3D-model could also be used to visualize the conjecture.

In real world applications, giving only definite answers to the membership questions could sometimes be too limiting. Since MAS allows us to use exact data only, we need to convert the user's indefinite answers to definite ones for the MAS algorithm to be able to use them. This conversion can be done completely transparently, but the user interface (especially a membership query dialog) needs to be modified to support multiple paths. The result of the synthesis would be a single nondeterministic statechart or multiple separate deterministic statecharts (depending on user's needs).

Currently, we have only limited experiences in using MAS. In fact, it has not been applied in any large real world application. For correctly directing the future development of MAS such experiences are essential. Therefore, case studies and gathering experiences form an important part of our future work.

Acknowledgements. The authors wish to thank Prof. Kai Koskimies for numerous useful discussions. This work was supported by TEKES, Nokia, Metso Automation, Plenware, Sensor Software Consulting, Ebsolut, and the Academy of Finland (Project 35025).

References

[1] Airaksinen J., Koskimies K., Koskinen J., Peltonen J., Selonen P., Siikarla M., Systä T.: xUMLi: Towards a tool-independent UML processing platform. *The Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER)*, 2002, to appear.

[2] Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75** (1987), 87–106.

[3] Balcázar, J.L., Díaz, J., Gavaldà, R., Watanabe, O.: Algorithms for learning finite automata from queries: a unified view. In D.-Z. Du, K.-I. Ko (eds.), *Advances in Algorithms, Languages, and Complexity*, Kluwer Academic Publishers, 1997, pp. 73-91.

[4] Birkendorf, A., Böker, A., Simon, H.U.: Learning deterministic finite automata from smallest counterexamples. In *Proc. 9th ACM/SIAM Symp. Discr. Alg. (SODA)*, January 1999, pp. 599–608.

[5] Chung, P., Huang, Y., Yajnik, S.: *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer.* [www-csag.ucsd.edu/individual/achien/cs491-f97/papers/dcom_corba.html]

[6] Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. *J. Comput. Syst. Sci.* **38** (1989), 86–124.

[7] Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science* **13** (2002), 5–51.

[8] Koskimies, K., Männistö, T., Systä, T., Tuomi, J.: Automated support for modeling OO software. *IEEE Softw.* **15** (1998), 87–94.

[9] Koskinen, J.: Implementing MAS on Windows NT (In Finnish). M.Sc. Thesis, Software Systems Laboratory, Tampere University of Technology, 2000.

[10] Koskinen, J., Mäkinen, E., Systä, T.: Minimally adequate synthesizer tolerates inaccurate information during behavioral modeling. In *Proc. of SCASE'01*, February 2001.

[11] Koskinen, J., Peltonen, J., Selonen, P., Systä, T., Koskimies, K.: Towards tool assisted UML development environments. In *SPLST'01*, Szeged, June 2001.

[12] Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In F.J. Ramming (ed.), *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, 1999, pp. 61–71.

[13] Leue, S., Mehrmann, L., Rezai, M.: Synthesizing software architecture descriptions from message sequence chart specification. In *Proc. of the 13th IEEE International Conference on Automated Software Engineering (ASE98)*, October 1998, pp. 192–195.

[14] Mäkinen, E., Systä, T.: MAS – an interactive synthesizer to support behavioral modeling in UML. In *Proc. of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, 2001, pp. 15–24.

[15] Mäkinen, E., Systä, T.: Minimally adequate teacher synthesizes statechart diagrams. *Acta Inform.* **38** (2002), 235–259.

[16] Mehlhorn, K.: *Data Structures and Algorithms 1: Sorting and Searching.* Springer, 1984.

[17] Microsoft Corporation: *COM (Component Object Model).* [http://msdn.microsoft.com/library/psdk/com/comportal_3qn9.htm], 2000.

[18] Microsoft Corporation: *Automation Start Page.* [http://msdn.microsoft.com/library/psdk/automat/autoportal_7145.htm], 2000.

[19] Microsoft Corporation: *COM Clients and Servers.* [http://msdn.microsoft.com/ library/psdk/com/comext_8p2r.htm], 2000.

[20] Microsoft Corporation: *Interfaces and Pointers.* [http://msdn.microsoft.com/ library/psdk/com/com_37w3.htm], 2000.

[21] OMG Corporation: OMG XML Metadata Interchange (XMI) Specification. [http://www.omg.org], 2000.

[22] The Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.3.1.* [http://www.omg.org/ technology/documents/formal/corba_2.htm], October 1999.

[23] Rational Software Corporation. *OMG Unified Modeling Language Specification v.1.3.* [http://www.rational.com], 2000.

[24] Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. *Inf. Comput.* **103**, (1993), 299–347.

[25] Schönberger, S., Keller, R., Khriss, I.: Algorithmic support for transformations in object-oriented software development. To appear in *Theory and Practice of Object Systems (TAPOS).*

[26] Selonen P., Koskimies K., Sakkinen M.: How to make apples from oranges in UML. In *Proc. of HICSS-34*, 2001.

[27] Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000, pp. 314–323.

[28] Wikman, J.: Evolution of a distributed repository-based architecture. In *Proc. of the First Nordic Workshop on Software Architecture*, Research Report 14/98, Dept. of Computer Science and Business Administration, University of Karlskrona/Ronneby, Sweden, 1998. [http://www.hk-r.se/fou/forskinfo.nsf/]

Acta Cybernetica 15 (2002) 567–581.

A distributed program synthesizer*

Vahur Kotkas†

Abstract

This paper describes an architecture of a distributed synthesizer for automated program construction. The objective of the synthesizer is to realize the ideas of Structural Synthesis of Programs in a computer network. The synthesizer handles structural specifications stored into Java classes as meta-interfaces and works on a network using CORBA technology.

Keywords: SSP, Java, Automated Synthesis of Programs, Meta-Interfaces.

1 Introduction

In the last decade Object Oriented programming (OOP) languages (like C++, Java, C#) became dominant providers of software reusability. However, the reuse efficiency greatly depends on developers' experience in programming and their knowledge on existing libraries.

There are several development environments available that assist in the selection of a proper libraries, but none of them generate fully operational code - this remains the developers' task. Searching for the proper library usually means reading the descriptions of the many libraries available. This is time consuming activity and hence software developers still create new libraries without knowing that some other software, having the same functionality, already exists. This indicates that there is a need for automated handling of software libraries.

One way to automate the software design process is to use Structural Synthesis of Programs (SSP) [1]. SSP is a technique of deductive synthesis of programs based on automatic proof search in intuitionistic propositional calculus. This technique uses classes of our problem domain extended with declarative specifications to generate new software automatically. The resulting software is correct (does not need any further checking) with respect to the correctness of the classes it is based on. The solving complexity is hidden from the end-user into the system.

The idea of using SSP for automated program generation is not new. Already in seventies a Priz family of programming languages was developed in the Institute of Cybernetics, that allow engineers to solve their tasks using a very high-level

*This work was partially funded by Estonian Innovation Foundation under the contract No. 6kl/00.

†Institute of Cybernetics at Tallinn Technical University, Estonia, email: vahur@cs.ioc.ee

programming language. A similar approach has justified itself quite well in the Amphion system [2].

Because of its relative robustness and flexibility the Java programming language was chosen for the SSP addition in the current study, but in principle this kind of additions can be provided to other OOP languages as well.

This paper introduces architecture of a software synthesizer that performs automated program construction and describes briefly the declarative specification language of the meta-interfaces added to Java classes that enable SSP.

This work is inspired from the work done in Institute of Cybernetics, Estonia during several decades and related to the work of Sven Lämmerman [3] from Royal Institute of Technology, Sweden.

2 Method

Java classes and objects do not contain sufficient information on their components relations and internal functionality to perform automated program synthesis. Automated program construction is not possible without that information. To overcome the problem, we introduce a meta-interface as an extension to a Java class, where the needed information is provided.

A meta-interface is a declarative specification that:

1. introduces a collection of interface variables of a class
2. defines, which interface variables are computable from others under which conditions.

For instance, having a class *Triangle* and a method *findSideSine* for computing size of a side according to the theorem of sine: $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$, we can introduce interface variables for all angles (A,B,C) and sides (a,b,c) of the triangle and declare a meta-interface that will specify how one can use the method. This meta-interface looks as follows:

```
var a,b,c,A,B,C : any;
rel a,A,B -> b {findSideSine}
rel b,B,C -> c {findSideSine}
 . . .
```

Here the *findSideSine* is an implementation of the theorem of sine in Java in the form of a method. The meta-interface just declares how the method can be used.

The usage of a meta-interface is as follows: one writes a request for synthesis of a method with input $x_1, \ldots, x_m$, where $m > 0$, and output y, whereas $x_1, \ldots, x_m, y$ are variables specified in the meta-interface, for instance in the form of the formula $x_1 \& \ldots \& x_m \rightarrow y$ and lets a prover to prove that this formula is derivable from the specification of the meta-interface, i.e. that the goal of synthesis of the requested method is achievable.

The prover returns a sequence of rules applied in the proof, which from the synthesis point of view represents an algorithm. This algorithm is used to generate the program code that solve the initial problem. Thus the algorithm is a co-result (or side-effect) of the proving process [4].

The aim of introducing meta-interfaces is obvious: to make classes as components more flexible. Indeed, if a meta-interface specifies n variables, then one can write 2^{2^n} requests for synthesis of which only a few may appear provable. However, we still get considerable flexibility compared to conventional interface of Java, without the need to implement all the possible programs that one may need during the runtime.

A meta-interface can be written for two different purposes. First, it may specify possible usage of the class, i.e. its derivable methods, like in our example. Second, it can be used as a specification showing how some application software should be composed from components that are supplied with meta-interfaces. In the latter case, a new class can be built completely from the specification of its meta-interface. For this purpose, specifying equality rules between some components of the new class may be needed.

3 Declarative specification language

The meta-interface, that contains declarative specifications, may be introduced to the classes

1. as an addition to the programming language
2. in the form of comments
3. as an array of strings.

The first solution, for adding the specifications, changes the Java programming language. The declarative specifications become native components of classes and are compiled with the rest of program. This avoids recompilations during the runtime. On the other hand, as we changed the language we need also a new Java compiler and virtual machine and we can not use our techniques with other versions of Java.

Introducing the structural specifications into a Java class as specifically formatted comments allows to use existing Java compilers and interpreters, as we do not need to modify existing programs — we just add some more comments to them. However, to find the specifications from the source files during the runtime is time consuming. Even more, this approach is not usable when source files may change or are not accessible during the runtime.

We have chosen the third possibility — to introduce the structural specifications into the class as an array of strings. In this case we can always access the specifications during the runtime as they are present in the component of the object in use. The Java programming language remains unchanged, hence, the solution

works with all versions of Java. However, this approach needs additional resources for compiling these specifications on the request of program synthesis.

The specification language of meta-interfaces consists of two sections — **var** and **rel** section. The **var** section specifies the components used in the **rel** section. Multiple instances of these sections can be used in a specification in random sequence.

The **var** section is an obligatory section in the specification of every class, without it the specification is incomplete. The **var** section is formally specified as follows

$$\texttt{var}\ a_1, a_2, \ldots, a_n\ :\ type,$$

where $a_i (i = 1..n)$ are declared variables. If *type* is represented with the keyword **any**, the declared component already exists in the Java class and the exact type of that component is applied during the compilation of the specification. Otherwise, if the names of Java primitive types or classes are used, new components are added to the synthesized program.

Some of the components in the **var** section may be defined as virtuals. In this case keyword **vir** is used instead of **var**. Virtual components are not taken into account when the state of the object they belong to is evaluated. This issue is further explained in the chapter 6.3.

The **rel** section defines relations of the declared components in the form of computability statements or computational constraints. These relations define usage of Java methods, equivalences, equations and inequalities. The statements are written as follows:

$$\texttt{rel}\ Label:\ RelStatement.$$

The *RelStatement* specifies an equivalence, equation, inequality or method declarations. The *Label* is a name given to the current specification statement that can be referred for debugging or is used for modifying inherited specifications.

As Java classes inherit properties from their superclasses, also the specifications of meta-interfaces are inherited. Meta-interface in a subclass overrides the specification statements of the superclass, if it defines a new relation with the same label.

Method declaration presents either a class method, an instance method or a special narrowing method denoted with the keyword **narrow**. It describes the input and output parameters required for the method invocation and exceptions if needed. A general method declaration construction is the following:

$$[InputSpec]\ \texttt{->}\ OutputSpec\ \{\ MethodName\ \}$$

Here the square brackets denote that the *InputSpec* is optional.

In case the *MethodName* is equal to keyword **narrow**, there should be exactly one component defined as method input and one for output. Keyword **narrow** represents a conversion of one type to another, if possible.

The *InputSpec* consists of two lists of components separated by &-symbol. The components described on both lists are separated by commas. The components of

the list before & are handled as method's formal parameters and the components after & respectively define instance variables that the method uses. If the list after the symbol & is empty, &-symbol must be discarded.

The *OutputSpec* has a similar structure to the *InputSpec*. The only difference is that before the &-symbol only one component is allowed, because an arbitrary method in Java may return only one value in time. In case there is no element specified before &, the method is of type void.

Additionally, the output parameter list may end with a | separated list of components that defines a set of exceptions, which could be thrown by the method.

For example `rel a, d.y & c.x, d.x -> c.y & d.y | e { doIt }` illustrates the usage of constructions, where *InputSpec* = `a, d.y & c.x, d.x` and *OutputSpec* = `c.y & d.y | e`. Component *a* and d.y are formal parameters for local method *doIt* with signature $type(c.y)\ doIt(type(a), type(d.y))$, and *c.y* is the output of that particular method. Global components *c.x* and *d.x* are used in the computations and *d.y* is modified as a side-effect of this method. The component e represents an exception that may be thrown by the method.

Equivalence defines a pair of components that should stay equal at any stage of an executed synthesized program. One can think of equvalent components as of objects that are stored into the same memory location. Equivalences are used as connectors between components enabling to build larger systems from smalles components. An example of an equivalence definition may be the following: `rel a.x == b.y`. One component may be present in many equivalence definitions. This forms a group of components that should stay equal during the execution.

Equation or inequation in the *RelStatement* is useful when one solves an engineering task. Java programming language does not include any solver that handle equations automatically and the solver for the equations have to be coded imperatively by the software developer. By allowing these kind of definitions, we can support constraint enriched Java classes and significantly reduce the programming time. However, we need a general purpose solver that handles these kind on specifications. Recent results in genetic algorithms show already today acceptable performance in optimization tasks, hence we are looking optimistically toward them.

There is a special use of having only one **var** component defined in the meta-interface — one can avoid from writing the full path to the subcomponent used in relations. For example instead of writing (`rel x = length.m + 7`) we can just write (`rel x = length + 7`), if the component *length* contains only one **var** component. In the synthesized program the component *length* would be automatically substituted to *length.m*, hence the computations go correctly. This allows to write the specifications in more convenient and more meaningful way.

Substitution is also useful when one does not know the name of the **var** component in the specific class, but is sure that there exists only one such component. This may happen when we prepare a superclass and the real types and components are defined in the subclasses.

4 Example of the meta-interfaces

Let us have triangulation as a sample problem. We have two triangles that have one side in common (see Fig. 1) and we have measured values for one side (a) and two angles (C and A) of the first triangle (t1) and two angles (C and A) from the second triangle (t2). Our task is to compute the total area of the two triangles.

Please note that the variable names on the figure do not denote the equivalence between different angles and sides — only the mapping between an object on the figure and its class is presented there.

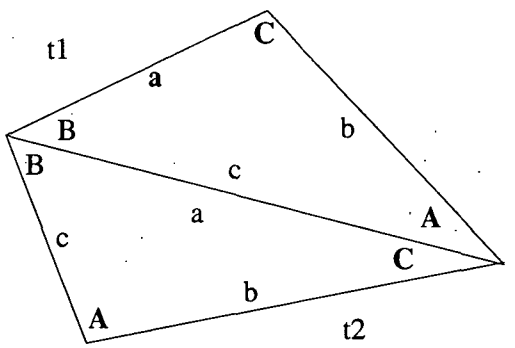


Figure 1: Graphical representation of the triangulation example.

While using an OOP language (like Java) we need to create two classes for the problem description. The first class describes a triangle and another composes the problem of triangulation from two triangles. For both classes we need to add a meta-interface (see Fig. 2 and the array of String SSPspec on Fig. 3) in order to specify the relations among the components.

```
var a,b,c,S : any
vir A,B,C : any
rel X: a,B,C -> S {calcArea}
rel Y: a,A,C -> c {findSideSine}
rel Z: A+B+C=Math.PI
```

Figure 2: Declarative specification of class Triangle.

All components (variables and objects) of the class (statements starting with var and vir), that are used in the relations (statements starting with rel), must be redefined in the specifications. Let us note that the actual type of components, specified here as of type any, is determined from the object during the synthesis.

Because of the encapsulation reasons private components can not be used in the declarative specification.

We can define also new components in the specification by giving their type instead of keyword any, but they do not become real components of the class. We may consider them as temporary components that are available only during the execution of the synthesized program.

The *calcArea* and *findSideSine*(see Fig. 2) are methods implemented in the class Triangle. The method *calcArea* realizes computation of the area, knowing one side and its two nearby angles. The method *findSideSine* computes a side, knowing another side and their opposite angles. The third *rel* statement describes relation among the angles of the class in the form of equation — the fact that the sum on inner angles of a triangle is equal to 180 degrees.

The first *rel* statement of class Problem (see Fig. 3) defines equivalence between the components. Equivalence means that the values of components stay always equal during the execution of the synthesized program.

```
import ee.ioc.cs.synthesizer.*;

public class Problem implements SSPInterface {
    public static String[] SSPspec = {
        "var t1, t2 : any",
        "var S : any",
        "rel U: t1.c == t2.a",
        "rel V: S = t1.S + t2.S"};
    public Triangle t1, t2;
    public double S;

    public void run() {
        t1.a = new Length('km',2);
        t1.C = new Angle('deg',90);
        t1.A = new Angle('deg',45);
        t2.C = new Angle('deg',45);
        t2.A = new Angle('deg',90);
        String progID = SSP.synthesize(this, "->S");
        SSP.execute(progID, this);
    }
}
```

Figure 3: The class Problem.

The labels X, Y, Z, U, V may be omitted. They may be used for debugging purposes, as they are added to the algorithm provided by the Planner. These labels are used here to make later in this paper references to these relations. The Planner and the algorithm are described in more detail in chapter 6.4.

As a result of the call *synthesize* of the class *SSP* (see the method *run*() on the Fig. 3) a new method will be synthesized (e.g uniquely identified as $xf17634$,

see Fig. 4) that realizes the requested computational problem. The name of the synthesized method is returned as its ID and can be used later for the method invocation.

The method *exec* executes the synthesized method and as a result modifies the object p by assigning proper value to the component S.

```
public void xf17634(Problem p) {
    p.t1.c = p.t1.findSideSine(p.t1.a,p.t1.A,p.t1.C);
    p.t2.a = p.t1.c;
    p.t1.B = Math.PI-p.t1.A-p.t1.C;
    p.t1.b = p.t1.findSideSine(p.t1.a,p.t1.A,p.t1.B);
    p.t1.S = p.t1.calcArea(p.t1.a,p.t1.B,p.t1.B);
    p.t2.c = p.t2.findSideSine(p.t2.a,p.t2.A,p.t2.C);
    p.t2.B = Math.PI-p.t2.A-p.t2.C;
    p.t2.b = p.t2.findSideSine(p.t2.a,p.t2.A,p.t2.B);
    p.t2.S = p.t2.calcArea(p.t2.a,p.t2.B,p.t2.C);
    p.S = p.t1.S + p.t2.S;
    return;
}
```

Figure 4: The synthesized method.

5 Distributed components

Software developers have long held the belief that complex systems are easier to be built from smaller components. There are two engineering drives in the development of a component-based system [5]:

1. Reuse - the ability to use existing components repeatedly
2. Evolution - development and maintenance of a highly componentisized system is easier and cheaper.

Composing a distributed application - an application composed of distributed components - adds the following features

1. Higher flexibility - as the components are not compiled into the application, the changes in the components do not affect the consistency of the distributed application if the interfaces of these components are fixed and semantics of their inputs and outputs remains unchanged.
2. Higher fault-tolerance - if a distributed component is developed for a certain environment and is well tested to work properly in it, then composing an application using these distributed components, that reside always in their native environment, cuts down in programming and testing time.

In a distributed application the intercomponent communication is fairly expensive in terms of time and other resources [6]. Thus components are encouraged to be larger than smaller. However, larger components have more complex interfaces and are changed more probably during the system development phase. The larger components are, the less flexible is the whole distributed application. To achieve optimum we should consider the level of abstraction of components, their likelihood to change, complexity etc. These ideas are followed while composing the architecture of the synthesizer, which is described in the next chapter.

6 Architecture of distributed synthesizer

By using the CORBA technology [7] in the synthesizer we get more flexibility for computing in a network and possibilities for concurrent computing, hence we call it a distributed synthesizer. Using CORBA also forces us to follow the ideas of modular programming and supports the program componentization.

While using CORBA for component interconnection we have to acknowledge that objects as entities can not be sent over a network, as CORBA is inter platform and inter programming language communication architecture. It means that every object should be serialized before delivery or in other words turned into a byte stream.

Serialization has two drawbacks on performance: firstly we need additional computing power for the serialization and secondly the serialized data takes always more memory than the original object, thus we may need wider bandwidth network for communication. This draws a more concrete granulation criteria to the decomposition process i.e. one should avoid loops in which there is a lot of data exchange over the network.

Considering the ideas described above we propose the following architecture for a distributed synthesizer, decomposed into 7 logically separated components (see Fig. 5): Object Factory, Compiler, Decorator, Knowledge Base (KB), Planner, Code Generator and Component Repository (CR)

6.1 Object Factory

The Object Factory (OF) is the central process that maintains the work of the other components. It does the performance evaluations on the network and sends a new task to the host that has least load at the moment. The Program uses a predefined local class SSP that finds the location of the OF on the current network and forwards the request to it. This allows to hide the whole machinery of the synthesizer from the end-user.

6.2 Compilation

Knowledge Base (KB) is a database-like structure that allows storing compiled declarative specifications for each class used in the program. When Compiler re-

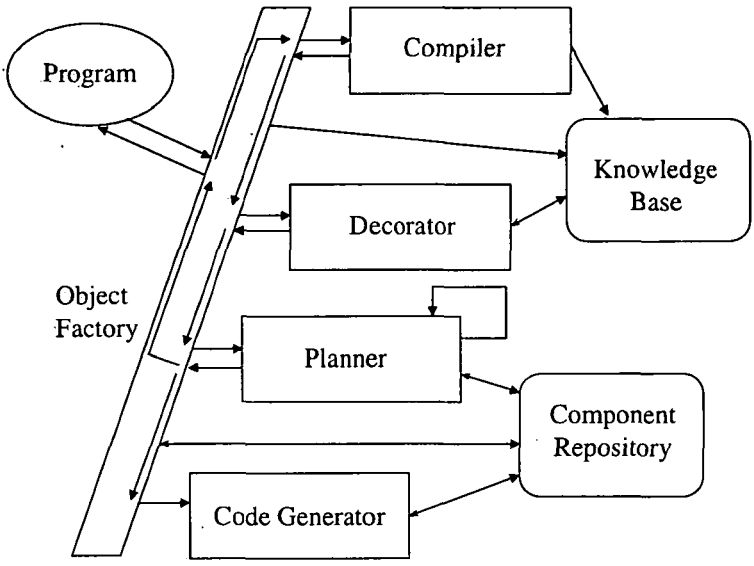


Figure 5: Architecture of the distributed synthesizer.

ceives a request for program synthesis from the OF, it first checks whether all the descriptions of classes that are used in the declarative specification are represented in the KB. The data received by the Compiler is consisting of serialized object, its serialized class and the computational problem specification i.e. the goal.

As we can never be sure that the classes stay unchanged the Compiler calculates a hash number based on the declarative specification for each class and compares it to the hash number stored in the KB. If the hash numbers are matching the compilation of that class can be skipped, otherwise if they do not match or the class description does not exist in the KB, the Compiler parses the declarative specifications of the class and stores the resulting description into KB.

Computing and checking the hash number assigned to class descriptions allows us to avoid compilation of the declarative specification every time a program synthesis request is made. Thus it speeds-up the program execution when similar problems have to be solved often.

6.3 Decorator

The Decorator creates a bipartite graph like structure out of the class descriptions stored in the KB. The bipartite graph has two types of nodes - components and relations [8]. The reason of building such a structure is to get rid of the object-oriented hierarchy, thus making it more suitable for the search. Fig. 6 presents

an example of a bipartite graph that corresponds to the sample specifications described above (see Fig. 2 and Fig. 3), where the smaller circles denote objects and components, and bigger circles represent to relations.

To still remember the hierarchy of the initial object, new relations are added to the graph (see unlabeled relation nodes on Fig. 5) that tie objects to their components.

From these additional relation is also clearly visible the distinction of the var and vir components of the declarative specification. Only the var components are included into the added relations and the vir components are left out. That means if there is a computational problem where the goal is an object having many components, then when all its var components are "known", meaning that the value of the component is known, the object gets the status "known" using that relation.

And vice versa - if an object is "known" the status transforms only to its var components.

The next task of the Decorator is to paint the object and variable nodes in the graph. A painted node in the graph represents to a component with the state "known". Considering our triangulation example the components a, C and A of Triangle t1 and components C and A of Triangle t2 are known, thus their nodes on the graph should be painted (dashed vertically on Fig. 6).

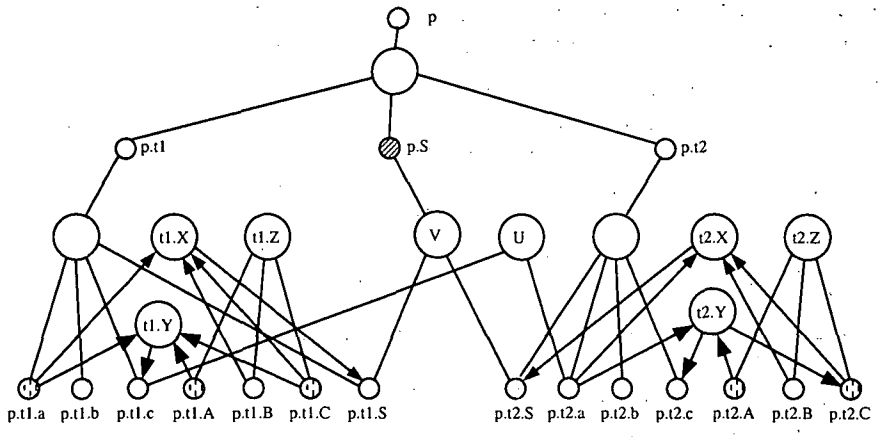


Figure 6: The search-space graph of the Triangulation example.

A different "color" is used to mark the goal. As the problem was to find the total area of the two triangles the node designating component p.S should be painted as goal (filled with upward diagonal lines on Fig. 6).

This structure is the search space delivered to the Planner.

6.4 Planner

The main function of the Planner is to find a solution for the problem specified by a user. Before starting with problem solving the Planner checks from the Component Repository (CR) whether a solution already exists for it. In the case the solution does not exist, the Planner starts solving it.

In principle there are two kind of relations that may occur in the declarative specification:

1. Unconditional relations implementing unconditional computability statements of SSP. In such relations computability of some (output) object depends only on some other (input) object(s). Unconditional relations of several types as equations, equivalencies, Java methods etc. are available in the specification language.

2. Conditional relations or relations with subtasks, implementing conditional computability statements of SSP, describe more sophisticated dependencies where output objects depend not only on input objects but also on solvability of some other computing problems. This kind of relations is unfortunately not present in our toy-example.

The problem specification for Planner is of form `x->y`, where `x` denotes the set of "known" (nodes painted with vertical dashes on the Fig. 6) objects and `y` denotes the set of objects to be computed (the goal). The Planner has to construct an algorithm (a sequence of relations) that describes how to compute y from x [4].

The proof search strategy of SSP applied in the Planner is

1. an assumption-driven forward search to select unconditional relations (linear planning). The algorithm works in the forward direction with unconditional relations only. At each step a relation, which input objects are "known" and at least one output object in "not known", is located and added to the algorithm. When adding a relation into the algorithm all its output objects are set as "known". The search is completed when all the nodes marked as "goal" are also "known" or there is no relations left with all inputs "known".

2. a goal-driven backward search to select and solve subtasks. The search is applied if the linear planning cannot be continued. Only such relations with subtasks are considered which input objects are "known". First the CR is checked for the existence of the solution of every subtask the relation have. If existing solutions are not found, the Planner is recursively used for solving every subtask of the relation considered. If all the subtasks of the relation are solved, the relation is added to the algorithm. Linear planning is used after every invocation of a relation with subtasks in the algorithm.

3. a minimization is applied to the resulting algorithm of the two previous search strategies. The search strategies above do not guarantee that we have built the shortest possible algorithm for computing the desired goal. Even more, the synthesized algorithm may contain relations that are not necessary for

computing the goal. Minimization is used to exclude such relations from synthesized algorithm. As a result of planning we get an algorithm that is not necessarily the shortest, but it does not contain unnecessary relations.

The algorithm is passed to the Code Generator and the problem specification in the CR is marked as solvable.

6.5 Code Generator

The code generation is a straightforward process, where the algorithm is translated into the class of the appropriate programming language i.e. to a Java class when the source language was Java. The class is compiled and a component including a newly created instance of this class is summoned and added to the CR. The component is stored into the CR with its problem specification that makes it possible to use that component repeatedly for solving many similar tasks.

If the computational problem were solvable and the algorithm is delivered to the Code Generator the Planner informs the OF about the solvability of the assigned problem and the OF delivers also the identification of the component stored in the CR to the Program.

Such an approach supports so-called Case Dependent Software Reuse (CDSR). It is called case dependent, because the planning process does not depend only on the declarative specifications given with particular object, but also on the current state of an object. The state of an object is determined by the states of its components (variables and objects), that can be marked as "known" or "not known".

6.6 Using synthesized components

When the distributed component is added to the CR, the user program may use it in two ways - by accessing the distributed component from the CR sending data to it and receiving output, or by retrieving the component's class and using its instance locally.

The aim of CR is to maintain a set of components implementing a certain interface. Thanks to the usage of this fixed interface we can use later all these components in program's initial context even when a certain component is not yet available at the moment of program creation.

An Object Factory that on a request creates the appropriate component and forwards the input parameters to it maintains the processing components of the distributed synthesizer. To take full advantages of the network, we may create multiple instances of the processing components on different servers and execute them in parallel.

Furthermore, the fault-tolerance of application is increased through redundancy provided by multiple instances of same component on different servers. Hence, if something happens with one server providing particular component, others backup it and we can distribute the system's load between different hosts in the network while solving different problems simultaneously.

We would not achieve any speed-up to the work of synthesizer when running just one user program, but the benefit appears when several user programs access the synthesizer simultaneously.

All components of the distributed synthesizer are implemented using the Object Management Group's Common Object Request Broker Architecture (OMG CORBA) [7], which provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing environments. CORBA enhances application flexibility and portability by automating many common development tasks such as object registration, location and activation; demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

Several aspects related to CORBA, Quality of Service, reliability of applications, and performance are studied in [9, 10, 11].

7 Conclusions

In the current paper meta-interfaces for Java classes are introduced and an architecture of a distributed program synthesizer is presented. The synthesizer is the main part of SSP realization that allows automatically construct programs out of declarative specifications provided by the meta-interfaces.

Our main objective is to extend a distributed programming language (like Java) with SSP capabilities that automates software reuse and helps users to design programs. The distributed synthesizer is a supporting tool that handles declarative specifications provided in the meta-interfaces of classes and does the actual program construction hidden from the end-user.

Here, an extension to the Java programming language is presented, but in principle such architecture of the synthesizer would suit also to other OOP languages, which classes are extended with structural specifications.

The efficiency of the synthesizer is low when considering only single user program execution, but the reuse of already synthesized programs gives a significant effect on the network where multiple agents are solving similar tasks.

For handling equation systems and additional constraints like inequations that enrich the specification language, a general solver has to be developed. We are looking optimistically towards genetic algorithms that have shown very promising results in solving different optimization tasks.

References

[1] E. Tyugu. The structural synthesis of programs, Lecture Notes in Computer Sciences, Vol. 122, 1981, pp. 290–303.

[2] M. Stickel, R.Waldinger, M.Lowry, T.Pressburger, I.Underwood. Deductive Composition of Astronomical Software from Suroutine Libraries. In 12th Con-

ference on Automated Deduction. A.Bundy, ed., Springer-Verlag Lecture Notes in Computer Science, Vol.814.

[3] S. Lämmermann. Automated Composition of Java Software, Lic. thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, Technical Report TRITA-IT AVH 00:03, ISSN 1403-5286, ISRN KTH/IT/AVH-00/03-SE, May 2000.

[4] M.Harf, E.Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, 6, 1980, pp 165-175.

[5] J. Hopkins. Component Primer. Communications of the ACM, October 2000, vol. 43, no. 10, pp. 27-30.

[6] D. Budgen, P. Brereton. Component-Based Systems: A Classification of Issues. Computer (IEEE CS), November 2000, vol. 33, no. 11, pp. 54-62.

[7] Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine, vol. 14, no. 2, February 1997.

[8] E.Tyugu, T.Uustalu. Higher-Order Functional Constraint Networks. Constraint Programming. NATO ASI Series F. Springer-Verlag 1994, pp 116-139.

[9] J. Zinky, D. Bakken, R. Schantz. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, vol. 3, no. 1, April 1997, pp. 1-20.

[10] S. Maffeis, D. C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine, vol. 35, no. 2, February 1997, pp. 56-60.

[11] A. Gokhale, D. C. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In Proceedings of GLOBECOM '96, London, England, November 1996, IEEE Press, pp. 50-56.

Acta Cybernetica **15** (2002) 583–599.

Automatic Test Selection based on CEFSM Specifications*

Gábor Kovács†, Zoltán Pap†, and Gyula Csopaki†

Abstract

Mutation analysis is a fault based testing method used initially for code based software testing. In this paper, this method is applied to formal specifications and used for automatic conformance test selection. This paper defines formally a set of mutation operators for CEFSM (Communicating Extended Finite State Machine) systems to enable the automated creation of mutant specifications. Mutants of a specification are used as selection criteria to pick out adequate test cases. Two different algorithms are proposed for the generation and selection of efficient test suites. Additionally, the operators and algorithms provide the basis of an automatic tool developed at the Budapest University of Technology and Economics. We present the results of an empirical study on the well-known INRES protocol acquired using the tool.

Keywords: automatic test generation, CEFSM, mutation analysis, SDL, test selection

1 Introduction

One of the most important criteria that applies to telecommunications software is compatibility with systems from different vendors. This is usually achieved by standardization. Manufacturers of the actual products ensure compatibility by applying these specifications. At the end of the development process, the final and most important step is to test the actual products to guarantee that they work as required by the specification. This is called conformance testing, which provides the means to ensure that systems from different companies are compatible, and are able to interoperate correctly according to the standard. The test development process, however, involves significant resources: it is very time consuming and requires the

*This research is supported by ETIK. ETIK partners are: Antenna Hungaria Co., Compaq Computer Hungary Ltd., Computer and Automation Research Institute of the Hungarian Academy of Sciences, Ericsson Hungary Ltd., Hungarian Telecommunication Co., KFKI Computer Systems Co., Sun Microsystems Inc., and Westel Mobile Telecommunications Co.

†Department of Telecommunications and Telematics, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117 Budapest, HUNGARY, Phone: (36) 1-463-2225, Fax: (36) 1-463-3107, e-mail: (kovacsg,pap,csopaki)@ttt-atm.ttt.bme.hu

manual effort of many well-trained developers. Therefore, its automation is an important challenge.

Specifications may be defined either in formal or informal way. The most widely accepted technique for formal specification of telecommunications protocols is SDL (Specification and Description Language) [8]. The SDL specification of a system is an excellent starting point for both manual and automatic test case generation. It describes the behavior of the system in detail, and its graphical interface provides easy readability. Even more importantly, its formal manner makes automation possible.

Previously, there have been attempts to automatically generate test cases from SDL specifications, but all of them face the common problem of test selection. All the different approaches require a mechanism to distinguish *"good"* test cases from the unnecessary ones, possibly without any human intervention. Thus, selection criteria are needed, that can be applied automatically.

The basic idea behind the recent method is that by applying small syntactical changes, or mutations, at atomic level to the specification exactly once at a time, we intentionally produce faults [10]. The rationale is that if a test set can distinguish a specification from its slight variants, the test set is exercising the specification adequately. The erroneous specifications then can be used as selection criteria to select adequate test cases.

Mutation analysis has previously been used for code-based software verification and validation, and it has also been applied to some simple specifications [2], for example SCR (Software Cost Reduction) [7] description of software. According to the literature [14][4][16], a similar idea has been used for protocol testing, called fault-based testing. Fault-based testing requires special fault models and tries to detect faults in the implementation, with respect to the specification.

In this paper, we define methods and mutation operators especially designed to enable the automation of test selection. We do not simulate typical errors of the specification or the implementation, instead we create erroneous specifications that provide an appropriate basis for the selection of test cases.

The paper is organized as follows. In Section 2, we define the CEFSM model formally, which describes the dynamic behavior of SDL processes. Section 3 is a summary on mutation analysis. Section 4 introduces mutation operators and approaches for test selection. In Section 5,we present the outcome of the empirical analysis of the presented method using the tool developed within the confines of this research. At last, a summary is given in Section 6.

2 Extended Finite State Machines

2.1 Specification and Description Language

SDL is a formal language, widely used for specifying – especially telecommunications – systems. It has been developed by ITU-T (CCITT). One of the strengths of SDL [8] is that it is a well-accepted world standard supported by ITU-T and ISO.

Nowadays, SDL is primarily used in the telecommunications industry for the description of telecommunication protocols during the development of different hardware structures and software products, but it can be used in other fields as well. Typically complex, event-driven, real-time, and communicating systems can be effectively described in SDL.

The static structure of the system is built up hierarchically with the system object as root. The system is the formal model of an existing or planned real system. Everything not belonging to the system is called environment. Below the system level are the blocks, which can build up several levels in a tree structure. At the bottom of the hierarchy are the processes. Communication between processes is possible via signals that travel on certain predefined routes connecting processes without delay.

In our case, the most important property of SDL is that it describes the dynamic behavior of a system as a CEFSM. Processes communicating via signals specify the operation of the system. An extended finite state machine describes each process. The state machines are labeled extended, since variables and timers can also be defined. All of the processes have their own memory for storing their variables and state information, and all of them contain a FIFO buffer of infinite length that is a queue for the incoming signals.

2.2 Formal Description of Communicating Extended Automata

Formally, a CEFSM, e.g. an SDL process or system, can be described by an quintuple [12] $CEFSM = (S,\ I,\ O,\ V,\ T)$, where S, I, O and T are the finite and nonempty set of states, inputs, outputs and transitions respectively, and V is the finite set of variables.

A transition $t \in T$ is a 6-tuple: $t = (s, i, P, A, o, s')$, where $s \in S$ is the start state, $i \in I$ is an input, $P : P(V)$ is a predicate on the variables, $A : V' := A(V)$ is an action on the variables, $o \in O$ is an output and $s' \in S$ is the next state.

Initially, the configuration of the machine is represented by the initial state $s_0 \in S$ and by the initial variable values.

Inputs ($i \in I$) and outputs ($o \in O$) are communication events. They may have parameters ($I \times V$ and $I \times V$), and are realized by parameterized signals in SDL. A reception of a parameterized signal can be viewed simply as an input and a joint action assigning the new values. The length of the queue increases by one as an input arrives and the input is added to the end of the queue. The length of the queue decreases by one as an input is processed. Henceforth, $i \in I$ denotes the input to be processed, which is not necessarily the first element. The SDL specific save mechanism means that if a specific input is the firts element of the queue, then it is skipped and the next element is going to be processed.

Variables provide further details of the system's internal state. The reaction to a specified input depends on the actual value of some variables through predicates. Predicates are expressions built up from the actual subset variables at a given state. Actions represent the effect of a transition to a subset of the variables.

A FSM (Finite State Machine) is reduced if it has no inaccessible states and all states are distinguishable [1]. A state s is inaccessible if there is no input sequence that moves the machine to state s. Two states s and s' of a machine M are distinguishable if there is some input sequence Q such that executing Q from s and s' produces different output. Let us define the reduced CEFSM by extending the definition of reduced FSM in [1]:

Definition 1 (Reduced CEFSM) *Variable $v_i \in V$ is inaccessible, if $\nexists i \in I$, that after a transition $t \in T$ $v \neq A(v)$. Variable $v_i \in V$ is observable, if there are different values $x \neq y$ of the variable v_i that if $v_i = x$ then the output is o_1 after the transition $t \in T$, and if $v_i = y$ then the output is o_2 after the transition $t \in T$, and everything else is unchanged, then $o_1 \neq o_2$.*

Boolean predicate $p_i \in P$ is inaccessible if $\nexists t_1, t_2 \in T$ that in the transition t_1 $p_i(V)$ is true, and in the transition t_2 $p_i(V)$ is false. Boolean predicate $p_i \in P$ is observable, if $\exists t \in T$, that if the output after the transition t is o_1 when $p_i(V)$ is true, and the output after the transition t is o_2 when $p_i(V)$ is false, and everything else is unchanged, then $o_1 \neq o_2$.

Let us say that a CEFSM M is reduced if no states, variables and predicates of M are inaccessible, any two states of M are distinguishable, and all variables and predicates of M are observable.

3 Mutation Analysis

Mutation analysis [13] is a white-box method for developing test cases – i.e. it is based on the knowledge of the internal logic of a system. Traditional mutation analysis checks for faults in the code of programs. We, however, apply mutation analysis to specifications instead of programs, and use it as selection criteria to pick out adequate black-box test cases.

A mutation analysis system defines a set of mutation operators [11], where each operator represents a type of an atomic syntactic change. Using these operators is practical for two reasons. On the one hand, they enable the formal description of fault types. On the other hand, operators make automated mutant generation possible. By applying the operators systematically to the specification, a set of mutants can be generated.

A mutation analysis system consists of three components (Figure 1):

- Original system.
- Mutant system – it is a small syntactic variation of the original. Mutants can be created by applying mutation operators, where each operator represents a small syntactic change.
- An oracle – a person or in most cases a program to distinguish the original from the mutant by their interaction with the environment.

Traditional program-based mutation analysis assumes the competent programmer hypothesis [3]. In the current work, we assume a similar *"competent specifier"*

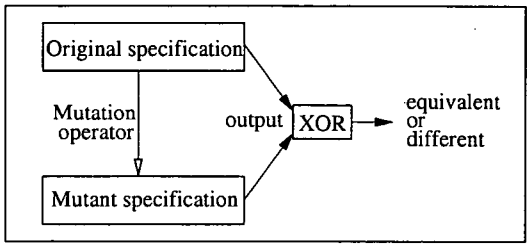


Figure 1: Components of a Mutation Analysis System

hypothesis stating that the specifier of a CEFSM system is likely to construct a specification close to the requirements, and so the test cases distinguishing syntactic variations of a specification are useful.

In the recent work, we only apply first-order faults – i.e. we apply exactly one mutation at a time –, because test sets detecting changes to the original system created by a simple error would also detect complex changes created by applying a sequence of simple mutations [13].

Test cases distinguish mutants from the original, if they produce different output. However, some of the mutants generated using the operators may be semantically equivalent to the original system. That is, a mutant and the original may compute the same function for all possible inputs. These mutants are called equivalent. All equivalents should be ignored, but all non-equivalents should be considered during test selection. Equivalence is a more complex problem in case of CEFSMs than in case of FSMs.

Definition 2 (Equivalence) *Let M_1 and M_2 be two CEFSMs with known structure, and identical input and output alphabet. Homeomorphism from M_1 to M_2 is the mapping Φ from S_1 to S_2 and the mappings Ψ_V, Ψ_P and Ψ_A from V_1 to V_2, P_1 to P_2 and A_1 to A_2 respectively, such that $\forall s_1 \in S_1$ and $\forall i \in I$ it is true for transition $t_2 = (\Phi(s_1), i, \Psi_P(P_1), \Psi_A(A_1), o_2, s_2'), t_2 \in T_2$ and transition $t_1 = (s_1, i, P_1, A_1, o_1, \Phi(s_1')), t_1 \in T_1$ that $s_2' = \Phi(s_1')$, $V_2' = \Psi_V(V_1')$ and $o_2 = o_1$.*

If Φ and Ψ are bijections, then it is an isomorphism. In this case, M_1 and M_1 are equivalent machines.

Definition 3 (Pseudo-equivalence) *Let M_1 and M_2 be two CEFSMs with known structure, and identical input and output alphabet. M_1 and M_1 are pseudo equivalent machines, if there exists a bijection Φ between S_1 S_2 such that $\forall s_1 \in S_1$ and $\forall i \in I$ it is true for transitions $t_2 \in T_2$ and $t_1 \in T_1$ that $o_2 = o_1$ if $s_2 = \Phi(s_1)$.*

Figure 2 shows the relationship among equivalents, pseudo-equivalents and mutants. We can make the following statements:

- The specification we investigate is an element of the set of equivalents $Spec \in EQ$.
- The set of equivalents is a subset of pseudo-equivalents $EQ \subseteq PE$.

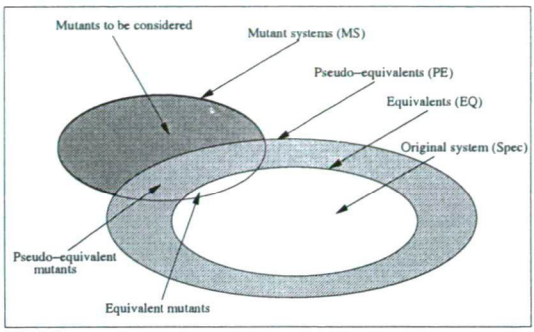


Figure 2: Equivalence Relationships (based on [16])

- A pseudo-equivalent mutant is an element of the $MS \cap PE$ set.
- An equivalent mutant is an element of the $MS \cap EQ$ set.

When applying the recent method, only mutant in the set $MS \setminus (MS \cap PE)$ should be considered.

4 Test Selection Method

4.1 Mutation Operators Proposed for CEFSM Specifications

In the recent paper, a very important consideration for the definition of operators is, that they should be simple enough to enable the automation of the mutation testing process, in order to design a tool. And additionally, the properties of SDL should also be taken into account. Operators should create finite – and as small as possible – number of mutant specifications. If possible, operators should not create any pseudo-equivalents and – of course – minimize the number of equivalents, because pseudo-equivalent systems unnecessarily increase the time required for the test case selection.

According to these considerations, mutation operators we use model atomic faults, and only one fault at a time. That is, we modify exactly one element in a transition $t \in T$. To select conformance test cases, it is essential to generate syntactically correct mutants. Syntactic correctness is necessary to be able to execute the mutant system. Some types of semantic errors should be detected, according to ([12]), since for example after applying an operator some states may become unreachable, preventing conformance testing. Such mutant systems should be detected and dropped during a semantic analysis phase.

We defined five classes of mutation operators for CEFSM descriptions according to which part of the automaton they are applied to:

- state modification operators,
- input modification operators,

- output modification operators,
- action modification operators,
- predicate modification operators.

Additionally, there is a specific operator for the mutation of save statements in SDL.

It is important to note, that we replace non-boolean predicates with a sequence of boolean predicates, and apply the operators on them. For the types of mutations when a component of the machine is replaced by an other component, instead of creating all possible combinations, it is sufficient to only produce one.

Henceforth, let the $\Omega()$ function represent the syntactical change applied.

Operator 1 (State) Modifying states. Here only the exchange of inputs should be considered. The mutation operator is applied either to the actual or to the next state:

1. Mutating the next state in transition $t \in T$: $t = (s, i, P(V), A(V), o, \Omega(s'))$. In this case we replace the next state, that is we lead the system to a wrong state and induce incorrect operation.
2. Mutating the actual state in transition $t \in T$: $t = (\Omega(s), i, P(V), A(V), o, s')$. This operator replaces the transition function of two states.

The mutation of the stating state is a special case of state mutation, where s_0 is modified: $\Omega(s_0)$.

Operator 2 (Input) The input mutation in the transition $t \in T$: $t = (s, \Omega(i), P(V), A(V), o, s')$.

1. Using $\Omega(i) := null$. This mutation is equivalent with the removal of a transition branch for an input at a given state.
2. Using the transition of input $i_x \in I'$, where $I' \subseteq I$ is the set of inputs, that have explicit transitions defined at the given s state. This means that we exchange the transition of two inputs.
3. Assigning the transition of input i_{inopp} ($i_{inopp} \in I$, but $i_{inopp} \notin I'$, where $I' \subseteq I$ is the set of inputs, that have explicit transitions defined at the given s) to the existing transition branch of input $i \in I$. This mutation means that we add a transition branch for the input i_{inopp}, that was implicitly consumed previously. Using this mutation, also the processing of inopportune (valid input arriving at wrong time) inputs can be inspected.
4. As mentioned previously, inputs and outputs may have parameters. If $i \in I \times V$, then we can mutate not only the input symbol, but the input parameter leaving the input symbol unchanged. This type of mutation is practically an action mutation, and can be viewed as the mutation of the implicit action assigning the new values. In this case, $\Omega(i) = i(\Omega(v))$, where $\Omega(v) := null$ – according to the action mutation (see below).

Operator 3 (Output) The mutation of an output event in the transition $t \in T$ is: $t = (s, i, P(V), A(V), \Omega(o), s')$. If the output symbol has parameters ($o \in O \times V$), then we have the possibility to modify the parameter: $\Omega(o) = o(\Omega(v))$, where $\Omega(v) := null$.

Operator 4 (Action) It is difficult to define a general mutation operator for actions, because of the presence of abstract data types. Only the $\Omega(A(V)) := null$ operator, that is the deletion of an action is suitable for this case. Missing action operator: $t = (s, i, P(V), \Omega(A(V)), o, s')$.

Operator 5 (Predicate) The mutation of boolean predicates has similar effects as the mutation of inputs.

1. Exchanging two branches of a decision in the transition $t \in T$ can be done simply negating the whole expression $t = (s, i, \Omega(P(V)), A(V), o, s'$, where $\Omega(P(V) := not(P(V))$.
2. Setting the predicate to be stuck-at-true ($\Omega(P(V)) := true$) or stuck-at-false ($\Omega(P(V)) := false$) brings on the removal of the other branch.

Operator 6 (Save) Since this paper consideres SDL to be the specification language used, we have to take its specialities, like the save mechanism (see Section 2.2), into account. Note that save is not part of the CEFSM model. This operator removes the save statement.

In Table 1, examples show some of the mutation operators and their realization in the context of SDL.

Operators	Original	Mutant
State	`NEXTSTATE wait;`	`NEXTSTATE connected;`
Input	`INPUT ICONresp;`	`INPUT IDISreq;`
Output	`OUTPUT CC;`	`OUTPUT DT (number, d);`
Action	`counter := 1;`	`/* Missing */`
Predicate	`DECISION sdu!id = CC;`	`DECISION NOT(sdu!id = CC);`
Save	`SAVE IDATreq (d);`	`/* Missing */`

Table 1: Mutation Operators for SDL

4.2 Theoretical Analysis of the Operators

Since the operators are defined in a formal manner, the analysis of their relationship is possible. We can show that there are correlations among the operators.

Theorem 1 (Relation between the brach exchange and the stuck at operator) *Applying the stuck-at (Ω_{SA}) and the branch exchange (Ω_{BX}) operators to the same predicate, then for the detection* $\Omega_{SA} \rightarrow \Omega_{BX}$.

Proof (based on [11]).
Let $p \in \mathcal{P}$ be the predicate to be mutated. The theorem follows: $\Omega_{SAfalse}(p) \rightarrow \Omega_{BX}(p)$ and $\Omega_{SAtrue}(p) \rightarrow \Omega_{BX}(p)$.
In case of deleting the true branch the detection condition is $p \oplus \Omega_{SAfalse}(p) = p \oplus false = p$.
In case of exchanging the branches the detection condition is $p \oplus \Omega_{BX}(p) = p \oplus not(p) = true$. That is, we always detect the branch exchange mutation. Since $p \rightarrow true$, then $\Omega_{SAfalse} \rightarrow \Omega_{BX}$.
In case of deleting the false branch the detection condition is $p \oplus \Omega_{SAtrue}(p) = p \oplus true = not(p)$. Since $not(p) \rightarrow true$, then $\Omega_{SAtrue} \rightarrow \Omega_{BX}$.
Therefore $\Omega_{SA} \rightarrow \Omega_{BX}$.

We made the following findings and statements concerning the operators:

- The mutation operator for states and next states are equivalent, since the tail state of a transition is the initial state of the next: $t_1 = (s, i, p(V), a(V), o, \Omega(s')) \Leftrightarrow t_2 = (\Omega(s'), i', p(V'), a(V'), o', s'')$.
- Timeouts in the CEFSM systems (e.g. SDL) can be considered simple inputs, and accordingly, the input mutation operator mutates them. To be able to test timer transitions, timeout events are made controllable from the environment. That is, whenever a test case reaches a timeout (for example `timeout T3;` in an MSC (Message Sequence Chart) test case), a corresponding input is sent directly to the owner machine of the timer from the environment, and after its consumption the corresponding timer transition is executed. During the test execution, in the test case a timeout explicitly indicates that the tester has to wait for the duration of the timer (e.g. "`Timeout T3`" and "Timer T3 shall be in the range 1 sec to 1.5 secs."). This way, methods for test case selection in the next section (4.3) become time independent.
- Definition 1 implies, that if using a reduced CEFSM, state exchange mutation and branch exchange for the predicates do not create equivalents, therefore, it is advised to use reduced specifications for mutant generation.

4.3 Algorithms for Test Generation

We defined two approaches for test generation. Although they are similar, being built on the same concept, they produce different resulting sets. Both approaches require the CEFSM specification of a system. There are practical and widely used tools, to assist the specification process like Telelogic Tau [15]. After the specification is completed, processing in both cases can be all automated.

We represent test cases in MSC (Message Sequence Chart) [9] [6]. The MSC test sets can later easily be transformed to different test description languages for example TTCN (Tree and Tabular Combined Notation). Tools support the semi-automatic TTCN table generation from MSC test case specifications.

Algorithm 1 (Deriving test cases from a specification) Figure 3 shows our first approach consisting of the following steps:

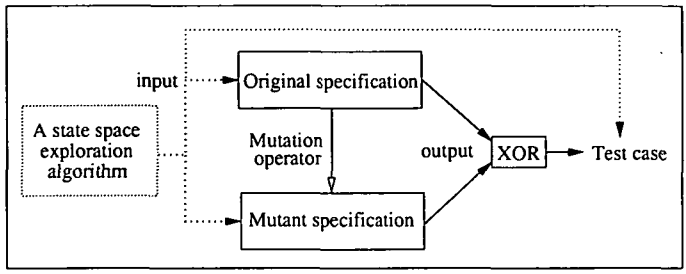


Figure 3: Mutation Analysis of CEFSMs Using a State Exploration Algorithm

1. Apply a mutation operator to the machine, that is, create a mutant specification.
2. Use a state space exploration algorithm to compare the mutant with the original system. Of course, we only stimulate the system using inputs from the environment, and only check the outputs to the environment for inconsistency. This is because our intent is to create test suites for black-box testing. In most cases, we should explore the state space of the original system, but in case of certain operators the desired result can be achieved by exploring the mutant state space. The algorithm must have break conditions, e.g. reaching a certain depth, exceeding a time limit etc.
3. When the state space exploration algorithm finds an inconsistency, it generates a test case based on the set of stimuli sent from the environment and the outputs received until the inconsistency was discovered.
4. Repeat steps 1-3 for all possible mutants until corresponding test cases are generated.
5. As a result we get a set of test cases.

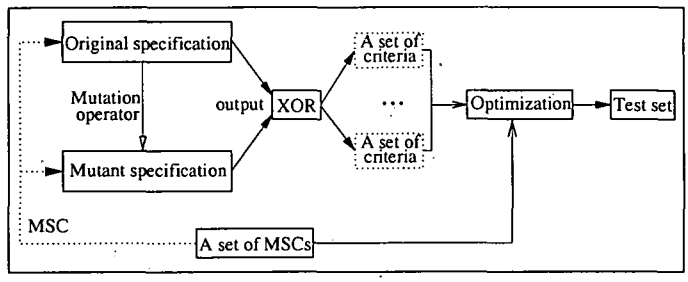


Figure 4: Mutation Analysis of CEFSMs Based on an Existing Test Set

Figure 4 shows our second scenario. Here we assume that a finite size, unstructured and highly redundant test set exists, for example in a form of a set of MSC test cases. These test sets can be created by the means of state space exploration

algorithms exploring the specification of the system. If we apply mutation operators checking inopportune inputs, this initial set of test cases also has to include inopportune test cases.

Let the matrix of criteria $\mathbf{C}$ be a two-dimensional matrix with boolean values.

Algorithm 2 (Selecting test cases from an existing set) Our second approach consist of the following steps:

0. Create a set of test cases.
1. Apply a mutation operator to the system, that is, create a mutant specification (the i^{th} mutant).
2. Run the test set on the mutant specification and check for inconsistency.
3. Create a row vector $\mathbf{C}_i$ (the i^{th} row of the C matrix).
   - Let $\mathbf{C}_{ij}$ be 0, if the j^{th} test case is not able to detect the i^{th} mutant.
   - Let $\mathbf{C}_{ij}$ be 1, if the j^{th} test case is able to detect the i^{th} mutant.

   In other words, select the test cases from the original set, which kill the given mutant.
4. Repeat steps 1-3 for i=1 to N, where N equals the number of all possible mutants that can be created from the given system using the mutation operators defined in Section 4.1.
5. Acquire the matrix of criteria, where rows represent the mutants and columns represent the test cases in the original set. This matrix describes for each test case the mutations the given test case is able to detect.
6. Apply some simplifications to the matrix of criteria (C):
   - If there is a column $\mathbf{C}_j$ in C, where $\forall i : \mathbf{C}_j[i] = 0$, it represents that the j^{th} test case did not find any of the mutants. Therefore, the j^{th} column can be omitted.
   - If there is a row C_i in C, where $\forall j : \mathbf{C}_i[j] = 0$, it represents either that the i^{th} mutant is an equivalent, or that there was no test case in the original set that could find the difference (kill the mutant).
   - If there are rows $\mathbf{C}_m$ and $\mathbf{C}_n$ in $\mathbf{C}$, where $\forall j : \mathbf{C}_m[j] \leq \mathbf{C}_n[j]$, then the row $\mathbf{C}_m[j]$ is unnecessary.
7. Select an optimal test suite from the original set, using an integer programming method.

We can also automatically assign weights to the test cases, implicating how time-consuming they are. Thus, we can further improve the efficiency of the selection.

Both algorithms have their advantages and drawbacks. The first algorithm requires less computation and time, but is it does not provide enough data for an optimization algorithm. An important benefit of the second – and more complex – method is that the optimization process ensures that only adequate test cases will

be selected, and thus, a more effective test set is created. A major drawback of the second method is that it is quite computation-intensive. If the two algorithms are executed consecutively, the first algorithm provides a rough set of – MSC – test cases, and then the second algorithm further improves the efficiency of the test set.

5 An experiment on the system INRES

5.1 The INRES system

We used the well-known sample telecommunications system INRES ([5]) to investigate the method presented (see Figure 5). We chose a sample protocol that includes all the typical properties of real life protocols. It is built on the OSI concept, and it contains some basic OSI elements. INRES is a connection-oriented protocol that operates between two protocol entities Initiator and Responder. These protocol entities communicate over a Medium service. Although it is not a real protocol, its specification contains most of the syntactic elements of SDL. The SDL specification of the system was created using the Telelogic Tau ([15]). The structure of the system and the states of the processes are shown in Figure 5. (The transitions of the processes are shown schematically, dots represent decisions. Inputs, outputs and actions are not represented. For more details see [5].)

Table 2 summarizes the syntactic properties of the six processes of the INRES system – described in SDL. The columns represent the following:

- *States* – number of states in the given process.
- *Inputs* – the sum of processed inputs in each state of the given process.
- *Outputs* – the sum of generated outputs in each transition of the given process.
- *Decisions* – the number of boolean decisions (i.e. the answer is either true or false) in all transitions of the given process.
- *Saves* – the number of save statements in the given process.
- *Transitions* – the number of different transitions in the given process.

Processes	States	Inputs	Outputs	Tasks	Decisions	Saves	Transitions
Initiator	4	10	9	8	4	1	14
CoderIni	1	3	4	5	2	0	5
Responder	3	5	7	2	1	0	9
CoderRes	1	4	3	6	1	0	5
MSAPMan1	1	2	2	0	0	0	2
MSAPMan2	1	2	2	0	0	0	2

Table 2: Syntactic Properties of the Processes in INRES

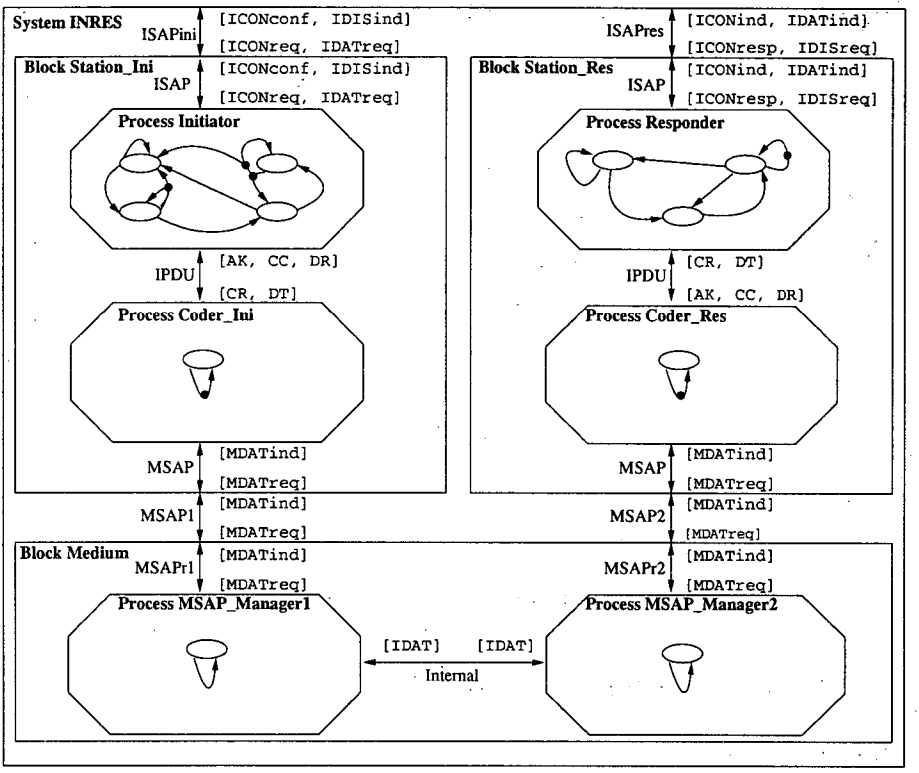


Figure 5: The INRES system in SDL

5.2 The tool

The tool consists of several components. One of the inputs of the tool is a textual SDL specification. First, this specification is modified, so that timers are made controllable, and decisions with multiple branches are transformed to series of boolean decisions. These transformations do not affect the behavior of the system. An other component of the tool applies mutation operators to this specification, and generates a set of mutant specifications. A semantic check is applied to this set to find critical semantic errors. The next component generates program code from both the original and the mutant specifications. The test execution component is the core of the tool. This implements the second algorithm. The inputs of this component are the program codes generated before and MSC test cases given in textual form. This component outputs a boolean matrix, which is the input of the last component, the optimizer. The final output of the tool the names of the MSC test cases found adequate.

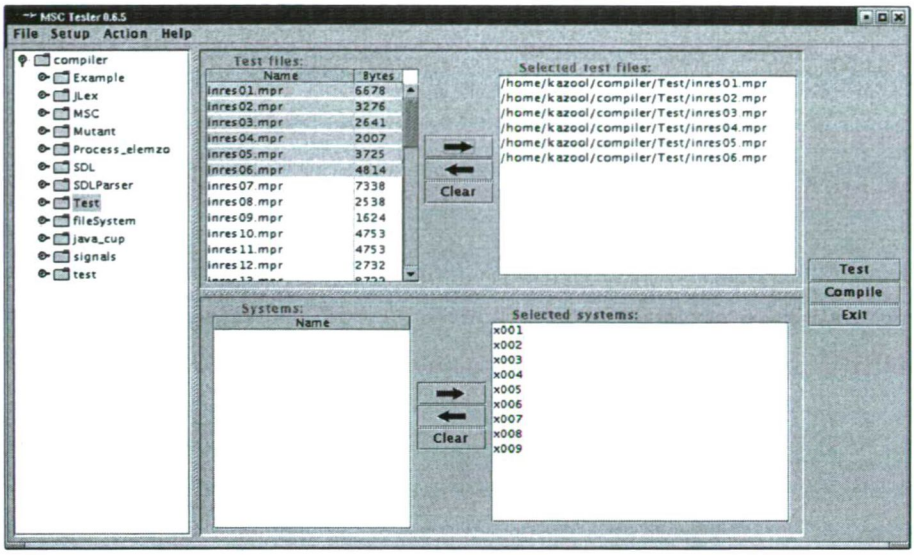


Figure 6: Mutation tester for SDL and MSC

5.3 The experiment and empirical analysis

We generated 87 MSC test cases using Telelogic Tau for the INRES protocol. This set intentionally included some appropriate and some useless test cases, and most of them were generated randomly. By applying the operators to the specification, 118 mutants were generated automatically. Finally, nine test cases of the original set were selected. The selection included some of the test cases we initially considered appropriate based on our knowledge of the system.

The resulting test set depends on the number and the quality of test cases in the original set. Therefore, if test cases in the initial set are generated automatically, then the resulting set is only influenced by their number. The selected test cases achieve 100% symbol coverage using the Telelogic Tau, which also indicates that they are, in fact, adequate.

This whole limited experiment took using the second algorithm on a computer with a PIII/500MHz processor and 256 Mbytes of RAM 103 minutes. The execution time – of both algorithms in Section 4.3 – can be decreased by dividing the test set and executing the groups in parallel. The time required to optimize an existing set – using the second algorithm – is proportional to the number and to the length of test cases and to the size of the specification.

Table 3 shows the data acquired during the experiment.

As the data show, twenty mutants have not been discovered by any of the MSC test cases. The worst mutation uncover ratio is, in the case of state mutation operators, and the best is in the case of input exchange. More than half (12) of the cases, where no difference has been found is in the Initiator process that is the

Mutation operators	Mutants generated	Detected	Detection ratio [%]
State	24	14	58
Input	28	28	100
Output	36	33	92
Action	21	15	71
Predicate	8	7	87
Save	1	1	100

Table 3: Mutation Analysis Applied to the SDL specification of INRES

largest in the system. Six of the mutants of the Responder process have not been killed. Both in case of processes CoderIni and CoderRes, one mutation has not been revealed.

There have been some mutations discovered by only one test case, we call them critical mutants. The test cases detecting these critical mutants have to be included in the resulting set. Interestingly, these test cases give eight of the nine selected. Out of the eight critical mutants, two have been generated using predicate, two using output, three using state and one using action mutation operator. Applying input mutation generated no critical mutants.

Both this, and the mutation uncover ratio indicate that input mutation (and input like mutations, e.g. missing transition) operators result in very rough mutant systems. That is, they produce radical changes in the behavior of the – INRES – system that can be detected by most of the test cases. State mutation, on the other hand, produces errors that can be discovered by only a small percentage of the test cases, but it has provided three critical mutants, more than any other operator.

6 Conclusions

Conformance testing is a vital part of the standard based telecommunications protocol development process. In the practice, the creation of test suites is usually a very time consuming manual process, even though several computer aided test generation methods have already been developed. By means of the method proposed, in this paper, it is possible to automate all steps of the test selection process, and to create effective test suites.

In this paper, we described how to apply mutation analysis, a white box method, to formal SDL specifications, and use mutant systems to automatically select adequate test cases for black box testing. For this purpose, we created and formally specified a set of mutation operators for CEFSM and SDL specifications. We also presented two slightly different algorithms for automatic test selection using the operators. We investigated empirically the mutation operators used. The recent method and the tool is useful not only for simple protocols, but also for real, complex telecommunications systems described in SDL.

In the future, we plan to make more experiments to reveal the effect of using different initial test and operator sets.

Acknowledgements

We would like to extend our gratitude to Zoltán Andriska, Gábor Bátori, Dung Le Viet and Antal Wu-Hen-Chang for their contribution in the tool development.

References

[1] A. V. Aho and J. D. Ullman. The Theory of Parsing, Translation, and Compiling, volume 1 of *Automatic Computation.* Prentice-Hall, 1972.

[2] P. E. Ammann and P. E. Black. *A Specification-based Coverage Metric to Evaluate Test Sets.* In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248, 1999.

[3] P. E. Ammann, P. E. Black, and W. Majurski. *Using Model Checking to Generate Tests from Specifications.* In *Second IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[4] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. *Automatic Test Generation for EFSM-based Systems.* http://citeseer.nj.nec.com/114451.html.

[5] J. Ellsberger, D. Hogrefe, and A. Sarma. SDL Formal Object-oriented Language for Communicating Systems. Prentice Hall, 1997.

[6] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. North Holland, 1993.

[7] K. L. Heninger. *Specifying Software Requirements for Complex Systems. IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

[8] ITU-T. *Recommandation Z.100: Specification and Description Language*, 1992.

[9] ITU-T. *Recommandation Z.120: Message Sequence Chart*, 1996.

[10] D. R. Kuhn. *A Technique for Analyzing the Effects of Changes in Formal Specifications. The Computer Journal*, 35(6):574–578, 1992.

[11] D. R. Kuhn. *Fault Classes and Error Detection in Specification Based Testing. ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[12] D. Lee and M. Yiannakakis. *Principles and Methods of Testing Finite State Machines – A Survey. Proc. of the IEEE*, 43(3):1090–1123, 1996.

[13] R. A. De Millo, R. J. Lipton, and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer*, 11(4):34–41, April 1978.

[14] Alexandre Petrenko, Gregor von Bochmann, and Ming Yu Yao. *On Fault Coverage of Tests for Finite State Specifications.* In *Computer Networks and ISDN Systems*, volume 29, pages 81–106, 1996.

[15] Telelogic Tau. http://www.telelogic.com.

[16] C.-J. Wang and M. T. Liu. *Generating Test Cases for EFSM with Given Fault Model.* In *INFOCOM 93*, volume 2, pages 774–781, 1993.

Acta Cybernetica **15** (2002) 601–620.

Resource-Conscious AI Planning with Conjunctions and Disjunctions*

Peep Küngas†

Abstract

The aim of this work is to develop a resource-conscious Artificial Intelligence (AI) planning system, which allows for nondeterminism in the environment. Such planner has a potential in applications where actions in "real world" are considered.

The planning process is based on proof search for a fragment of Linear Logic (LL) [10] sequents using a subset of LL rules. As LL is resource-conscious and has additive disjunction connective (represents nondeterminism), LL sequents are used to describe an application domain, whereby every LL sequent represents pre- and postconditions of a particular action execution.

We present an idea to use Petri net reachability tree analysis for finding proofs for propositional LL sequents. Game playing is used to solve LL additive disjunctions. From LL proofs plans are extracted which because of underlying LL properties keep track of resources and handle both deterministic and nondeterministic actions.

Keywords: Linear Logic Theorem Proving, AI Planning, Petri Nets, Game Playing.

1 Introduction

In mission critical situations the response of a system must be reactive. For a system relying on a symbolic representation it means that a backup plan, describing alternative actions to be carried out if the main plan is not applicable anymore, should be available immediately. Therefore nondeterminism in results of actions should be taken into account already in the planning phase and action representation must thus support describing nondeterministic actions.

The aim of this work is to develop a resource-conscious AI planning system, which is able to manage nondeterminism in an environment. Such planner has a potential in applications where actions in the "real world" are considered.

*This work was partially supported by the Estonian Science Foundation under grant no. 4155.

†Software Department, Institute of Cybernetics at Tallinn Technical University, Akadeemia tee 21, 12618 Tallinn, Estonia, e-mail: peep@cs.ioc.ee.

There exists a framework called Linear Logic [10] (LL) which allows handling uncertainties while being also resource-conscious.

As LL is resource-conscious, we are using LL formulae to describe application domain and LL theorem proving to construct a plan achieving a goal. We propose a subset of LL connectives and operators sufficient for describing both deterministic and nondeterministic actions in a resource-sensitive world.

For proving propositional LL sequents, we present a new approach by composing Petri net reachability tree analysis and game playing. Plans are then extracted from these proofs. Some extensions and improvements to our algorithm are described in [19], where also the algorithm is compared to other systems and algorithms in the AI planning field.

2 Linear Logic

LL is a refinement of classical logic introduced by J.-Y. Girard to provide a means for keeping track of "resources"—two assumptions of propositional constant A are distinguished from a single assumption of A. Although LL is not the first attempt to develop resource-oriented logics (well-known examples are relevance logic and Lambek calculus), it is by now the most investigated one.

Since its introduction LL has enjoyed increasing attention both from proof theorists and computer scientists. Therefore, because of its maturity, LL is useful as formal representation of planning system kernel. Good tutorials to LL are [32] and [21]. One of the first overviews of LL applications is presented in [1]. There exist several efficient formal method tools for proving LL sequents [31].

From the complete set of LL connectives and operators we are using multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$) and "of-course" (!). Whilst the connectives $\otimes$ and $\oplus$ are needed to describe pre- and postconditions of actions, the operator ! gives us control over resources.

In terms of resource acquisition the logical expression $A \otimes B \vdash C \otimes D$ means that the resources C and D are obtainable only if both A and B are obtainable. Thus the connective $\otimes$ defines deterministic relations between resources and actions.

The expression $A \vdash B \oplus C$ on the contrary means that if we have a resource A, we can obtain either a B or a C, but we do not know which one of those. It is definitely clear that $\oplus$ is suitable to represent nondeterminism in results of actions.

The operator ! means that we can use or generate particular resource as much as we want—the resource is somehow unlimited for us.

To illustrate the above let us consider the following LL formula, adapted to our set of LL connectives and operators, from [21]—$(D \otimes D \otimes D \otimes D \otimes D) \vdash (H \otimes C \otimes !F \otimes (P \oplus I))$, which encodes a fixed price menu in a fastfood restaurant: for 5 dollars (D) you can get an hamburger (H), a coke (C), all the french fries (F) you can eat plus a pie (P) or an ice cream (I) depending on availability.

To increase the expressiveness of formulae, we are using the $a^n = \underbrace{a \otimes \ldots \otimes a}_{n}$, for $n \geq 0$, with the degenerate case $a^0 = 1$.

Since we do not use all LL connectives and operators for planning, only a subset of LL rules listed below is needed for proof search.

Logical axiom and Cut rule:

$$A \vdash A \quad (Axiom) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \ (Cut)$$

Rules for the propositional constants:

$$\vdash 1 \qquad \frac{\Gamma \vdash A}{\Gamma, 1 \vdash A}$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta} \ (L\oplus) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(a) \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(b)$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta} \ (L\otimes) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'} \ (R\otimes)$$

Rules for the exponential !:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (W!) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (L!) \qquad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma, \vdash !A, ?\Delta} \ (R!) \qquad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (C!)$$

3 Planning and LL

One merit of LL deductive planning is said [11] to consist in its ability to solve the technical frame problem [24] without the need to state frame axioms explicitly and is therefore especially good for representing causal relations between actions and resources.

The multiplicative conjunction connective ($\otimes$) and additive disjunction ($\oplus$) have been used in [23], where a demonstration of robot planning system has been given. The usage of ? and !, whose importance to AI planning is emphasised [4], is discussed there, but not demonstrated.

Influenced by [23], LL theorem proving has been used by Jacopin [14] as an AI planning kernel. As only the multiplicative conjunction $\otimes$ is used in formulae there, the problem representation is equivalent to presentation in STRIPS [9]-like planners—the left side of a LL sequent represents STRIPS *delete*-list and the right side accordingly *add*-list. Multiplicative conjunctions just separate propositions.

Unfortunately, the algorithm Jacopin proposes for proof search is very inefficient and belongs to the class of brute force methods.

In [6] a formalism has been given for deductively generating recursive plans in Linear Logic. This advancement is a step further to more general plans, which are capable of solving instead of a single problem a class of problems.

To illustrate the usefulness of LL in resource-aware planning we give proofs for the two following tasks. After that, according to the LL rules and axioms used, plans from the proofs are extracted. Application domains are represented as sets of extralogical axioms.

It should be mentioned that as the LL planning idea consists finding a proof for a certain LL sequent, the goal for the planner is specified with a LL sequent: *initial conditions* $\vdash$ *final conditions*. In order to implement plan reuse, every proved sequent (theorem) can be added to the set of extralogical axioms and other previously proved sequents, describing a particular application domain. Thus the next time it can be used to prove other sequents.

The LL sequents we are going to use for describing application domains and goals are given in form $D \vdash D$ whereas $D ::= K \mid K \oplus D$, $K ::= 1 \mid A \mid K \otimes A$ and A is an atomic formula.

The first task for a robot is to mine a ton of gold and the second is to fill a box with balls.

3.1 Gold-mining problem

Let us assume that in some particular case the application domain consists of three actions: Excavate, Refine and Convert. We have also two predicates—KG and T for inspecting whether we have a kilogram or a ton of gold respectively. At last two constants—*Sand* and *Gold*—are used to indicate the resources we are mining.

The STRIPS-like *add-* and *delete*-effects for the just mentioned actions are specified with LL sequents as follows:
Excavate: $\vdash KG(Sand)$
Refine: $!KG(Sand) \vdash KG(Gold)$
Convert: $KG(Gold)^{1000} \vdash T(Gold)$

It should be reminded that the *delete*-effect of an action is defined before the $\vdash$ symbol and *add*-effect after that symbol.

The proof for the sequent $\vdash T(Gold)$ follows here[1]:

$$\dfrac{\dfrac{\dfrac{\vdash KG(Sand)}{\vdash !KG(Sand)}\,(R!) \qquad !KG(Sand) \vdash KG(Gold)}{\vdash KG(Gold)}\,(Cut)}{\vdash KG(Gold)^{1000}}\,999 \times (R\otimes) \qquad KG(Gold)^{1000} \vdash T(Gold)}{\vdash T(Gold)}\,(Cut)$$

The plan $\mathcal{P} = \{1000 * \{n * \mathbf{Excavate}, \mathbf{Refine}\}, \mathbf{Convert}\}$ extracted from the previous proof could be stated in the natural language as: excavate sand by one kilogram and extract gold from it until you have a kilogram of gold. Repeat in such a way thousand times.

[1] Literally speaking—how to get a ton of gold from nothing.

3.2 Box filling problem

In the second example task a robot has to fill a box with balls. The robot can move around and pick up balls it encounters on its way. Every ball is then stored in a box. The box is said to be full if it holds seven balls. The robot starts moving with empty hands (propositional constant $EMPTY$). The goal in LL sequent form is $EMPTY \vdash EMPTY \otimes FULL$.

The expression system consists of five propositions. The specification of actions available for the second task is listed here:

Take: $EMPTY \otimes NEAR \vdash HOLD$
Store: $HOLD \vdash EMPTY \otimes IN$
Move: $\vdash NEAR$
Fill: $IN^7 \vdash FULL$

First we prove that the sequent $EMPTY \vdash EMPTY \otimes IN$ is derivable:

$$\dfrac{\dfrac{\dfrac{EMPTY \vdash EMPTY \quad \vdash NEAR}{EMPTY \vdash EMPTY \otimes NEAR}\,(R\otimes) \qquad EMPTY \otimes NEAR \vdash HOLD}{EMPTY \vdash HOLD}\,(Cut) \qquad HOLD \vdash EMPTY \otimes IN}{EMPTY \vdash EMPTY \otimes IN}\,(Cut)$$

The plan extracted from the proof is $\mathcal{P}_1 = \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}$. The preceding proof was used to construct a plan for finding and storing a ball. The following generates a plan to fill a box with balls found:

$$\dfrac{\dfrac{EMPTY \vdash EMPTY \otimes IN \qquad \dfrac{\dfrac{\dfrac{EMPTY \vdash EMPTY \quad IN \vdash IN}{EMPTY \vdash EMPTY \otimes IN^6 \quad IN \vdash IN}\,R\otimes}{EMPTY, IN \vdash EMPTY \otimes IN^7}\,(L\otimes)}{EMPTY \otimes IN \vdash EMPTY \otimes IN^7}\,(Cut)}{EMPTY \vdash EMPTY \otimes IN^7} \qquad \dfrac{\dfrac{EMPTY \vdash EMPTY \quad IN^7 \vdash FULL}{EMPTY, IN^7 \vdash EMPTY \otimes FULL}\,(R\otimes)}{EMPTY \otimes IN^7 \vdash EMPTY \otimes FULL}\,(L\otimes)}{EMPTY \vdash EMPTY \otimes FULL}\,(Cut)$$

The plan $\mathcal{P}_2 = \{7 * \mathcal{P}_1, \mathbf{Fill}\}$ presents another plan, which uses a previously canned plan $\mathcal{P}_1$. In this way a modular plan representation is achieved and plan reuse is implemented.

In plans using deterministic actions it is always clear which actions must be executed to achieve a goal. The situation is more complex, if we include nondeterministic actions to an application domain specification. In that case the plan must cover all cases possibly occurring because of nondeterministic actions. A plan is valid if execution of every action in its sequence leads to a goal.

Although LL has been demonstrated to be useful for AI planning [23, 14, 11, 4, 6], there has been little discussion about which algorithms to use for proving LL sequents.

It is intuitively clear that for a smaller set of logic connectives, operators and rules simpler proving techniques are applicable. Thus it makes sense to look for new proving methods.

4 Petri nets and LL

It has been shown [7, 3] that Petri nets can be presented in the form of LL sequents. Thus at least a part of a set of LL sequents can be translated into Petri nets. One of the first surveys on Petri nets is [28], where an overview of basic concepts and extensions and subclasses of Petri nets may be found.

4.1 Petri nets

A Petri net is a formal tool which is particularly well suited for representing true parallelism, concurrency and causal relations in discrete event dynamic systems. In this section we define the concept of Petri net and give the main notation and definitions to be used in the sequel.

A Petri net is a 5-tuple $N = (P, T, Pre, Post, M_0)$, where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $Pre : P \times T \to \mathcal{N}$ is the input incidence function, $Post : T \times P \to \mathcal{N}$ is the output incidence function and $M_0 : P \to \mathcal{N}$ is the initial marking. A Petri net with a given initial marking is denoted by (N, M_0).

In the graphical representation, circles denote places and vertical bars denote transitions, tokens are represented as dots inside places. The *Pre* incidence function describes the oriented arcs connecting places to transitions. It represents for each transition t the fragment of the state in which the system has to be before the state change corresponding to t occurs. $Pre(p, t)$ is the weight of the arc (p, t), $Pre(p, t) = 0$ denotes that the place p is not connected to transition t.

The *Post* incidence function describes arcs from transitions to places. Analogously to *Pre*, $Post(t, p)$ is the weight of the arc (t, p).

The vectors $Pre(., t)$ and $Post(t, .)$ denote all input and output arcs respectively of transition t with their weights.

The Petri net dynamics is given by firing enabled transitions, whose occurrence corresponds to a state change of the system modeled by the net. A transition t is enabled for a marking M, if $M \geq Pre(., t)$. This enabling condition is equivalent to $\forall p \in P, M(p) \geq Pre(p, t)$. Only enabled transitions can be fired.

If M is a marking of N enabling a transition t, and M' is the marking derived by the firing of a transition t from M, then $M' = (M - Pre(., t)) + Post(t, .)$. The firing is denoted as $M \xrightarrow{t} M'$.

In a Petri net N it is said that a marking M_g is reachable from a marking M iff there exists a sequence of transitions s such that $M \xrightarrow{s} M_g$. We call the *reachability problem* for Petri nets the problem of finding a firing sequence s to reach a given marking M_g from M_0.

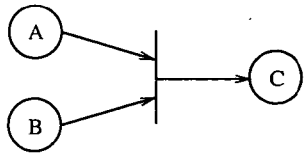


Figure 1: The Petri net representation of LL sequent $A \otimes B \vdash C$.

The *coverability problem* (sometimes also called the submarking reachability problem), given a marking M_g, is defined as the problem of finding a firing sequence s to reach a marking M_s from M_0 such that $M_g \subseteq M_s$.

4.2 Mapping LL sequents to Petri nets

LL sequents involving only $\otimes$, like $A \otimes B \vdash C \otimes D$, can be presented directly in form of Petri nets. Then the set of places P of a Petri net N is augmented with places A, B, C and D. The set of transitions T is augmented with a new transition t_i. *Pre* and *Post* are augmented respectively with $Pre(A, t_i)$, $Pre(B, t_i)$ and $Post(t_i, C)$ plus $Post(t_i, D)$.

Using LL rule $L\oplus$ the sequent $A \oplus B \vdash C$ is splitted into sequents $A \vdash C$ and $B \vdash C$. Constructions like $!A$ may be used only in the left hand side of an sequent, which has to be proved (a goal sequent).

The semantics of the connective $\oplus$ on the right hand side of a sequent should be implemented explicitly using other techniques. The Petri net representation of a LL sequents containing multiplicative conjunctions is demonstrated in Figure 1.

It must be noted that the left hand side of a goal sequent forms the initial state M_0 (marking) of a Petri net and the right hand side forms the final Petri net state M_g (goal in AI planning terminology) which must be achieved as a result of proof search. Thus a proof is found if there exists a way to fire Petri net transitions so that from the initial state the final state is reached.

4.3 Rewriting "of-course" in formulae

To fit into the Petri net framework, formulae containing the "of-course" operator must be rewritten. The following rules should be kept in mind, when doing that:

1. there may be no ! in extralogical axioms
2. there may be no ! in the right hand side of a sequent for which a proof has to be generated
3. for every $!X$ in the left hand side of that sequent generate a new extralogical axiom $\vdash X$ and remove $!X$ from left hand side of the initial sequent
4. if there are several instances of $!X$, then only one axiom $\vdash X$ is generated and all instances are removed from the left side of the sequent

For instance the LL sequent $!A \otimes C \vdash B$ to be proved is translated to sequents $\vdash A$ and $C \vdash B$, whereas $\vdash A$ is a new extralogical axiom and $C \vdash B$ the new sequent to be proved.

5 Solving LL multiplicative conjunctions with Petri nets

Petri nets have been used in AI planning for example in [26, 5, 30, 25] thanks to their clear and well-defined semantics, as well the formal analysis techniques and tools available.

There exist several other works considering AI planning with graphs—quite similar by ideology to Petri nets. For example in [12] colouring of bipartite graphs is used in goal search.

Another AI planner using graph as a planning structure is Graphplan [2], where the application domain, initial conditions and goals are presented as nodes and arcs between nodes. Graphplan uses breadth-first search for finding a solution for achieving a goal. A plan in Graphplan is represented in partial order.

Kanovich [16, 17] proved that the derivability problem of the LL subset consisting only of tensor $\otimes$, modal storage operator $!$, and linear implication $\multimap$ is directly equivalent to Petri net reachability problem and thus is decidable.

While using Petri net reachability tree analysis for theorem proving, many irrelevant choices in proof search are eliminated, making proof search tractable by:

- avoiding useless loops, which are generated for instance by applications of the *Cut* rule ,
- reducing the set of permutations of inference as applications of some inference rules like $L\otimes$ and $R\otimes$ are ignored

Useless loops in a proof are characterised by the following situation, where one sequent σ inside a proof is identical to the root of the proof:

$$\begin{array}{c} \vdots \\ \dfrac{}{\sigma} \\ \vdots \\ \dfrac{}{\sigma} \end{array}$$

Hence, the sub-proof starting from that internal sequent could replace the overall proof.

5.1 Petri net reachability tree analysis

For analysing properties of Petri nets, the basic technique used involves finding a finite representation for the reachability set of a Petri net. The representation used is known as the *reachability tree*, which consists of a tree whose nodes represent

markings (states) of a Petri net and whose arcs represent the possible changes in Petri net state resulting the firing of transitions.

Thus a reachability tree represents all Petri net states reachable from an initial Petri net state using all of its transitions.

However, the reachability set of a marked Petri net is often infinite. Thus to form a finite representation of an infinite set we must map many markings into the same node of the tree. This mapping is accomplished by *collapsing* a set of states into a state by ignoring the number of tokens in a place of the net when this number becomes "too large". This is represented by using special symbol ω. The symbol ω represents a value which can be arbitrarily large (infinite), whereby $\omega + a = \omega$, $\omega - a = \omega$ and $a < \omega$, where a is an arbitrary positive integer.

Each node in the reachability tree is labelled with a marking, arcs are labelled with transitions. The initial (root) node is labelled with the initial marking. Given a node x in the tree, additional nodes are added to the tree for all markings that are directly reachable from the marking of the node x. For each transition t_j which is enabled in the marking for node x, a new node with marking[2] $\delta(x, t_j)$ is created and an arc labelled t_j is directed from the node x to this new node. This process is repeated for all new nodes.

Continuing this process will obviously create the entire reachable state space. A path from the initial marking to a node in the tree corresponds to an execution sequence. Since the state space may be infinite, two special steps [18] are taken to define a finite reachability tree.

First, if a new marking is generated, which is equal to an existing marking on the path from the root node to the new marking, the new (duplicate) marking becomes a terminal node. Since the new marking is equal to the previous marking, all markings reachable from it have already been added to the reachability tree by the earlier identical marking. Detecting a new duplicate marking is used later in this article also under the term *cycle detection.*

Second, if any new marking x is generated, which is greater than a marking y on the path from the root node to the marking x, then these components of marking x, which are greater than the corresponding components of marking y are replaced by the symbol ω (this action is further called *collapsing*). Since marking x is greater than marking y, any sequence of transition firings which is possible from marking y, is also possible from marking x. In particular, the sequence that transformed marking y into marking x can be repeated indefinitely, each time increasing the number of tokens in those places, which have a ω. Thus the number of tokens in these places can be made arbitrarily large. A sequence of labels of arcs from the node y to the node x, would be referred later with term *subplan.*

As an example of this construction, consider the marked Petri net in Figure 4. The Petri net consists of 6 places—($EMPTY$, $NEAR$, $HOLD$, IN, $MOVE\_OK$, $FULL$) and 4 transitions—(**Take**, **Store**, **Move**, **Fill**). Initially we assume that places $EMPTY$ and $MOVE\_OK$ both hold one token.

[2] δ is a transition function from one Petri net state to another.

Thus, the initial state of that Petri net is coded as $\{1,0,0,0,1,0\}$, where the number at the first position is the number of tokens at place $EMPTY$, the second position corresponds to the number of tokens at place $NEAR$, the third at $HOLD$, the fourth at IN, the fifth at $MOVE\_OK$ and the sixth at $FULL$.

We begin with $\{1,0,0,0,1,0\}$ as the root node of the tree. In this marking we have only one enabled transition—**Move**. Thus we have a new node corresponding to firing **Move**, $\{1,1,0,0,0,0\}$ and an arc from $\{1,0,0,0,1,0\}$ to $\{1,1,0,0,0,0\}$. From this marking we can fire **Take** and from that newly created node then **Store** resulting in $\{1,0,0,1,1,0\}$. Now, since $\{1,0,0,1,1,0\} > \{1,0,0,0,1,0\}$, we replace the forth component by ω. This reflects the fact that we can fire sequence $\{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}$ arbitrary number of times and make the number of tokens in the place IN as large as desired.

From the marking $\{1,0,0,\omega,1,0\}$ we can fire transitions **Fill** and **Move**. After firing **Fill** again collapsing takes place because $\{1,0,0,\omega,1,0\} < \{1,0,0,\omega,1,1\}$. It can be seen that after firing sequentially **Move**, **Take** and **Store** from marking $\{1,0,0,\omega,1,0\}$ we reach again marking $\{1,0,0,\omega,1,0\}$, which is a duplicate and therefore is set to terminal node. The partial Petri net reachability tree is as shown in Figure 5.

Although the Karp-Miller algorithm is useful for the analysis of Petri net reachability tree, due to the loss of information caused by ω, it cannot detect all possible firing sequences needed. A workaround for that problem and several interesting examples about Karp-Miller algorithm may be found at [33].

5.2 Representation of application domain, valid plans and goals

An application domain specified with LL sequents is translated to a Petri net using previously defined transformations. For disjunctions special nodes are added, where splitting to different Petri net places is done.

A valid plan and a subplan is represented by a sequence of the Petri net transitions to be fired for achieving a goal. Every transition may refer to a subplan, which has to be applied after that transition zero or more times. The number of repetitions is computed while checking the correctness of the plan. Note that subplan in our case is not a plan for achieving subgoals—it is just a reusable sequence of transitions.

Every transition in the plan and subplan is enriched with its precondition, which is presented by a Petri net state where that transition was fired. None of transitions in subplan can refer to another subplan. Every subplan is enriched with its precondition. Goal is presented by a Petri net state.

5.3 The PNSolver algorithm

The depth-first algorithm for using Petri net reachability tree analysis for generating plans, where only deterministic actions are considered, is presented in Figure 2.

Naturally other search methods can be easily adapted to that algorithm.

Initially the list of passed states consists of just the initial Petri net state. Initial plan *plan* is empty, *goal* is the state which must be achieved while firing transitions ("executing" actions).

The main idea of the algorithm is to test all transitions in the set T whether they are fireable from a certain Petri net state or not. If a transition is fireable, a new state is computed and it is checked whether the state already exists in the

```
Algorithm PNSolver(init, proof, H, goal)
inputs:  init //initial state of a Petri net
         proof //an initial proof
         H //a set of states visited during the proof search so far
         goal //a final state of a Petri net
output:  P //a set of valid proofs

begin
for ∀t ∈ T
    state ← init
    if fireable(t, state) then
        state ← fire(t, state)
        if state ∈ H then //that state has been visited
            continue //do not proceed (Karp-Miller)
        end if
        proof.push(t)
        state ← collapse(state, H) //collapse state space (Karp-Miller)
        H.push(state)
        if state ≡ goal then //a proof is found
            announce(proof)
            proof.pop()
            H.pop()
            continue
        end if
        proof.pop()
        H.pop()
    end if
end for
for ∀p ∈ announcedProofs()
    p ← CheckCorrectness(init, p, goal)
    P ← P ∪ p
end for
return P
end PNSolver
```

Figure 2: A pseudocode for finding sequences of transitions from the initial state of a Petri net to the final state.

sequence of visited states H. If the state happens to exist in H, a cycle is detected and search from that Petri net reachability tree node is terminated. As the cycle was detected, it is clear that if the goal state was not found in previous round of that cycle, it would not be found on the next round either.

Else, if the *state* was not discovered in H, *state* is added to H, *plan* is augmented with a transition, and Petri net possibly infinite state space is collapsed, if possible. Collapsing generates a subplan, which is added to a list of subplans and a reference from the last transition in the plan to that subplan is inserted. Also information about how many resources that subplan generated, consumed and its precondition is remembered. For more information about Petri net reachability tree analysis see [28].

The usage of subplans reduces dramatically the time needed to construct a plan if used wisely [29]. In our case subplans are generated as a side-effect using Petri net state space collapsing.

If the achieved state is equivalent to *goal*, search is terminated at that particular reachability tree node, plan is added to a list of plans P, and the inspection of next transitions begins.

In the case goal is not achieved after firing particular transition, search from the new state is recursively proceeded until whole available search space is explored. The possibly infinite search-space is reduced by cycle detection and collapsing.

Checking the correctness of a plan (see algorithm in Figure 3) in the end of algorithm is started to solve ambiguities generated through collapsing. Correctness checking computes how many times certain subplans must be executed sequentially to achieve needed amount of resources. During the correctness checking it may turn out that some goals are not valid at all and the exact number of some resources cannot be achieved. It may turn out for example that only even number of units of a resource may be generated instead of needed odd number defined by the goal.

Correctness checking starts from the goal state and moves towards the initial state, while undoing effects of fired transitions. If a transition referring to a subplan is detected, the number of subplan execution cycles is computed according to needed resources. If finally Petri net state *init* is achieved, the plan is considered to be valid and is returned.

To illustrate this algorithm, let us take a look again at the box filling problem (see Sect. 3.2) we solved previously and modify[3] the application domain specification to be more "real":

Take: $EMPTY \otimes NEAR \vdash HOLD$
Store: $HOLD \vdash EMPTY \otimes IN \otimes MOVE\_OK$
Move: $MOVE\_OK \vdash NEAR$
Fill: $IN^7 \vdash FULL$

The Petri net representing the same specification consists of 6 places ($EMPTY$, $NEAR$, $HOLD$, IN, $MOVE\_OK$, $FULL$) and 4 transitions (**Take**, **Store**, **Move**, **Fill**). Initially we assume that places $EMPTY$ and $MOVE\_OK$ both have one

[3] In the previous version of the specification a robot was able to run from one place to another and then pick up balls it encountered from one final place.

```
Algorithm CheckCorrectness(init, proof, goal)

inputs:  init //initial state of a Petri net
         proof //an initial proof
         goal

output:  p //a complete proof

begin
state ← goal
for ∀t ∈ p
    if referenceToSubsolution(t) then
        t ← addSubsolutionRepetitions(t, state, init)
        state ← undoSubsolution(t, state)
    else
        state ← undo(t, state)
    end if
end for
if state ≡ init then
    return p
else
    return nil
end CheckCorrectness
```

Figure 3: A pseudocode for checking the validity of a proof.

token and the goal would be to get one token to each of $EMPTY$, $MOVE\_OK$ and $FULL$. In LL terms it means finding a proof for a sequent $MOVE\_OK \otimes EMPTY \vdash FULL \otimes MOVE\_OK \otimes EMPTY$. See also Figure 4 for graphical representation of that Petri net and its initial state.

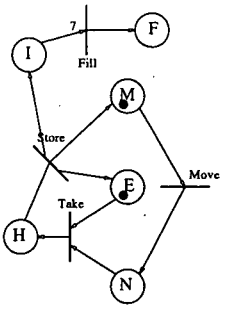


Figure 4: The Petri net with an initial marking for the box filling example.

So, the initial state of that Petri net is coded as $\{1,0,0,0,1,0\}$, where the number at the first position is number of tokens at place $EMPTY$, the second position corresponds to the number of tokens at place $NEAR$, the third at $HOLD$, the fourth at IN, the fifth at $MOVE\_OK$ and the sixth at $FULL$. The goal state is accordingly $\{1,0,0,0,1,1\}$. Petri net reachability tree for our application domain and goal specification is in Figure 5.

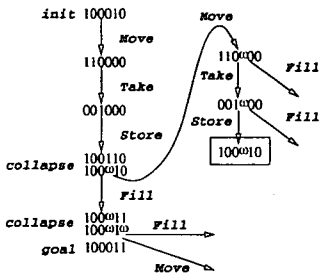


Figure 5: A fragment of reachability tree of Petri net in Figure 4. Arcs without ending node show that the tree follows.

Petri net transitions can be presented with STRIPS-like *add-* and *delete*-lists, see Table 1. The precondition in STRIPS sense for every transition is that there is enough of tokens to fire particular transition.

Transition	*delete*-list	*add*-list
Move	000010	010000
Take	110000	001000
Store	001000	100110
Fill	000700	000001

Table 1: Petri net transitions as *add-* and *delete*-lists.

It can be seen from the reachability tree that after firing transitions **Move**, **Take** and **Store**, we reach a state where compared to the initial state the number of tokens at place IN has increased: $\{1,0,0,1,1,0\} > \{1,0,0,0,1,0\}$. Therefore this component is replaced by ω, meaning that using that sequence again we can generate infinite number of resources IN—this is called Petri net state space collapsing. After collapsing we apply **Fill** and reach the state which is a possible goal state and resulting plan is $\mathcal{P} = \{n * \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}, m * \mathbf{Fill}\}$.

It is evident that through Petri net state-space collapsing we reduce the search space, but we lose information about how many times a subplan must be executed. Therefore we have to start with correctness checking to compute the number of times we have to execute generated subplans. In that particular case the final plan is $\mathcal{P} = \{7 * \{\mathbf{Move}, \mathbf{Take}, \mathbf{Store}\}, 1 * \mathbf{Fill}\}$.

Empirically estimating, a valid plan with our algorithm (even when using breadth-first search) may be found with smaller number of steps than Graphplan planner could, because in our case subplans are used and cycles are detected while planning.

Unfortunately finding the *shortest* plan requires searching the whole available search space—all valid plans are computed, subplans are unfolded and then the shortest plan is selected. It is due to the fact that we do not know how many times a subplan must be executed—it may be 0 as well as 1000 times. As the previous example illustrated, a plan with length 22 was found from the Petri net reachability tree at depth 4. Thus there is no strong connection between a search depth and a plan length if the Karp-Miller algorithm is used.

6 Using game playing for solving LL additive disjunctions

In respect to planning, having a LL sequent $A \vdash B \oplus C$, we do not know which one of the resources B or C would become the result of execution of particular action. Therefore a way to handle sequents like $A \vdash B \oplus C$ within Petri nets we may choose between stochastic Petri nets [27] and coloured Petri nets [15], sometimes misleadingly called high-level nets. In such a way the model can be specified within one net. The colour of tokens depends on which disjunct was selected as a result of applying particular transition.

As stochastic Petri nets alter in some way the firing rule of Petri nets and make it so the main part of net theory no longer applicable [27], they should be avoided as long as possible.

Instead of using previously mentioned Petri net derivations we start game playing on a Petri net reachability tree. The advantage of game playing on a tree is pruning of search space by using AND nodes. Thus transition selection represents OR and disjunct selection represents AND level of a game.

An advanced PNSolver algorithm, PNGameSolver, for solving disjunctions on the right hand side of LL sequents is presented in Figure 6. The only difference with algorithm in Figure 2 is that here reachability of the goal state from all disjuncts is considered. If at least from one disjunct the goal is unachievable, search with particular transition is terminated, backtracking is performed and search at other OR node proceeds.

If the goal at least from one disjunct is not achievable, plans for other disjuncts at the same AND node are discarded. In addition, it must be noted that at the current moment the effect of subplans including nondeterministic actions is not quite clear and therefore Petri net state space collapsing is not applied if at least one action in resulting subplan would represent a nondeterministic action. Therefore generating plans including disjunctions is quite exhaustive.

```
Algorithm PNGameSolver(init, proof, H, goal)
output:  P //a set of valid proofs

begin
upper_cycle:  for ∀t ∈ T //OR node
    for ∀disj ∈ t //AND node
        ...
        if notAchievable(disj) then
           continue upper_cycle
        end if
    end for
end for
...
return P
end PNGameSolver
```

Figure 6: A pseudocode for handling nondeterminism within Petri nets.

7 Computational complexity of the PNSolver and the PNGameSolver algorithm

Lipton proved [22] an exponential space lower bound for Petri net reachability problem, while the known algorithms require nonprimitive recursive space. As PNSolver algorithm uses Petri net reachability tree analysis for finding solutions, its minimal complexity is EXPSPACE-hard.

PNGameSolver uses additionally games in Petri net reachability tree analysis, which makes its complexity comparable to reachability problem of nondeterministic Petri nets, whose complexity is proved [16] to be undecidable.

However, tight complexity bounds of the reachability problem are known for many Petri net classes [8]. For example, If we would limit expressive power of used LL sequents so that sinkless or normal Petri nets may be used, we would achieve NP-complete complexity [13] for reachability analysis.

As we are using Petri net reachability tree analysis *on the fly*, meaning that we build Petri net and analyse it simultaneously, it is possible to bypass difficulties arised from complexities of algoritms by using heuristics at search. If powerful heuristics is used only a fragment of reachability tree would be generated before a solution is found.

Another constraint, we may set, is to fix a bound on the number of tokens at places—algorithms for bounded Petri nets are less expensive in complexity.

In [20] an abstraction technique for LL theorem proving is proposed which in the best case reduces the exponential problem solving complexity of PNSolver algorithm to linear.

8 Conclusions

In this article we proposed a new way of resource-conscious AI planning using propositional LL sequents as a knowledge representation form. These sequents are translated into Petri nets and then a plan achieving a certain goal is computed.

A fusion of Petri net reachability tree analysis and game playing is used to solve problems described with LL sequents. Whilst game playing allows handling LL additive disjunctions, Petri net analysis handles LL multiplicative conjunctions and preserves resource-consciousness.

By Petri net state space collapsing subplans are generated and thereby plan generation time is reduced.

Experimental results with PNSolver algorithm are presented in [19], where a comparison between depth-first and breadth-first search algorithms with and without certain extensions is given.

Acknowledgements

I would like to express my gratitude to Tarmo Uustalu from Software Department, Institute of Cybernetics at Tallinn Technical University for his assistance in LL.

Also I would like to thank Keijo Heljanko from Theoretical Computer Science Laboratory, Helsinki University of Technology for introducing me to the computational complexity of Petri net problems and pointing to some weaknesses and faults in my algorithms.

References

[1] V. Alexiev. Applications of Linear Logic to Computation: An Overview. Bulletin of the IGPL, Vol. 2, No. 1, March 1994.

[2] A. L. Blum, M. L. Furst. Fast Planning Through Planning Graph Analysis. Artificial Intelligence, Vol. 90, pp. 281–300, 1997.

[3] C. T. Brown. Linear Logic and Petri Nets: Categories, Algebra and Proof. PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1991.

[4] S. Brüning, S. Hölldobler, J. Schneeberger, U. Sigmund, M. Thielscher. Disjunction in Resource-Oriented Deductive Planning. Technical Report AIDA-93-03, Technische Hochschule Darmstadt, Germany, 1994.

[5] S. Caselli, F. Zanichelli. On assembly sequence planning using Petri nets. Proceedings of IEEE International Symposium on Assembly and Task Planning, Pittsburgh, Pennsylvania, August 1995, pp. 239–244, 1995.

[6] S. Cresswell, A. Smaill, J. Richardson. Deductive Synthesis of Recursive Plans in Linear Logic. In Proceedings of the Fifth European Conference on Planning, pp. 252–264, 1999.

[7] U. Engberg, G. Winskel. Petri nets as models of Linear Logic. In: A. Arnold (ed). Proceedings of Colloquium on Trees in Algebra and Programming (CAAP'90), Copenhagen, Denmark, May 15–18, 1990, Lecture Notes in Computer Science, Vol. 431, Springer-Verlag, pp. 147–161, 1990.

[8] J. Esparza, M. Nielsen. Decidability Issues for Petri Nets—a Survey. Journal of Information Processing and Cybernetics, Vol. 30, No. 3, pp. 143–160, 1995.

[9] R. Fikes, N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, Vol. 2, pp. 189–208, 1971.

[10] J.-Y. Girard. Linear Logic. Theoretical Computer Science, Vol. 50, pp. 1–102, 1987.

[11] G. Grosse, S. Hölldobler, J. Schneeberger. Linear Deductive Planning. Journal of Logic and Computation, Vol. 6, pp. 232–262, 1996.

[12] M. Harf, E. Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, Vol. 6, pp. 165–175, 1980.

[13] R. Howell, L. Rosier, H. Yen. Normal and Sinkless Petri Nets. Journal of Computer and System Sciences, Vol. 46, pp. 1–26, 1993.

[14] E. Jacopin. Classical AI planning as theorem proving: The case of a fragment of Linear Logic. In: AAAI Fall Symposium on Automated Deduction in Nonstandard Logics, Palo Alto, California, AAAI Press, pp. 62–66, 1993.

[15] K. Jensen. Coloured Petri Nets. In: W. Brauer, W. Reisig, G. Rozenberg (eds). Petri Nets: Central Models and Their Properties. Proceedings of Advances in Petri Nets 1986, Part I, Bad Honnef, September 8–19, 1986, Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, pp. 248–299, 1987.

[16] M. I. Kanovich. Petri Nets, Horn Programs, Linear Logic, and Vector Games. In: M. Hagiya, J. C. Mitchell (eds). Theoretical Aspects of Computer Software, International Symposium TACS'94, Sendai, Japan, Lecture Notes in Computer Science, Vol. 789, Springer-Verlag, pp. 642–666, 1994.

[17] M. I. Kanovich. The complexity of Horn fragments of Linear Logic. Annals of Pure and Applied Logic, Vol. 69, No. 2–3, pp. 195–241, 1994.

[18] R. M. Karp, R. E. Miller. Parallel program schemata. Journal of Computer and Systems Sciences, Vol. 3, No. 2, pp 147–195, May 1969.

[19] P. Küngas. Linear Logic Programming for AI Planning. Master thesis, Tallinn Technical University, Estonia, Research Report CS 103/02, Institute of Cybernetics at Tallinn Technical University, May 2002.

[20] P. Küngas. Linear Logic Theorem Proving with Abstraction. To appear in Proceedings of 14th European Summer School in Logic, Language and Information, ESSLLI'2002, Trento, Italy, 5-16 August, 2002.

[21] P. Lincoln. Linear Logic. ACM SIGACT Notices, Vol. 23, No. 2, pp. 29–37, Spring 1992.

[22] R. J. Lipton. The Reachability Problem Requires Exponential Space. Department of Computer Science, Research Report 62, Yale University, 1976.

[23] M. Masseron, C. Tollu, J. Vauzeilles. Generating plans in Linear Logic I–II. Theoretical Computer Science, Vol. 113, pp. 349–375, 1993.

[24] J. McCarthy, P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer, D. Michie, M. Swann (eds). Machine Intelligence, Vol. 4, Edinburgh University Press, pp. 463–502, 1969.

[25] Y. Meiller, P. Fabiani. Planning with Petri nets. Proceedings of RJCIA'2000, 2000.

[26] K. E. Moore, A. Gungor, S. M. Gupta. Disassembly process planning using Petri nets. Proceedings of IEEE International Symposium on Electronics and the Environment, Oak Brook, IL, pp. 88–93, 1998.

[27] A. Pagnoni. Stochastic Nets and Performance Evaluation. In: W. Brauer, W. Reisig, G. Rozenberg (eds). Petri Nets: Central Models and Their Properties. Proceedings of Advances in Petri Nets 1986, Part I, Bad Honnef, September 8–19, 1986. Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, pp. 460–478, 1987.

[28] J. L. Peterson. Petri nets. ACM Computing Surveys, Vol. 9, pp. 223–252, 1977.

[29] D. Ruby, D. Kibler. Learning Subgoal Sequences for Planning. Proceedings of IJCAI'89, Detroit, Michigan USA, 20–25 August 1989, Vol. 1, pp. 609–614, 1989.

[30] F. Silva, M. Castilho, L. A. Künzle. Petriplan: a new algorithm for plan generation (Preliminary report). In M. C. Monard, J. S. Sichman (eds). Advances in Artificial Intelligence. Proceedings of International Joint Conference 7th Ibero-American Conference on AI 15th Brazilian Symposium on AI, IBERAMIA-SBIA 2000, Atibaia, SP, Brazil, November 19-22, 2000, Lecture Notes in Computer Science, Vol. 1952, Springer-Verlag, 2000.

[31] T. Tammet. Proof Strategies in Linear Logic. Journal of Automated Reasoning, Vol. 12, pp. 273–304, 1994.

[32] A. S. Troelstra. Tutorial on Linear Logic. In: P. Schroeder-Heister, K. Dosen (eds). Substructural Logics, Oxford University Press, pp. 327–355, 1993.

[33] F.-Y. Wang. A Modified Reachability Tree for Petri Nets. Proceedings of 1991 IEEE International Conference on Systems, Man, and Cybernetics, Blackburg, VA, pp. 329–334, 1991.

Acta Cybernetica **15** (2002) 621–632.

Experiences in Modelling Feature Interactions with Coloured Petri Nets

Louise Lorentsen*, Antti-Pekka Tuovinen† and Jianli Xu†

Abstract

A modern mobile phone supports many features: voice and data calls, text messaging, personal information management like phonebooks and calendars, WAP browsing, games, alarm clock, etc. All these features are packaged into a handset with a small display and a special purpose keypad. The limited user interface and the seamless intertwining of logically separate features cause many interactions between the software components in the UI of mobile phones. In this paper, we present an overview of our approach to modelling feature interactions in Nokia's mobile phones with explicit behavioral models of features. We use Coloured Petri Nets as the modeling language and the tool Design/CPN that provides a graphical, interactive user interface for constructing and simulating Coloured Petri Nets. We describe at a general level how we have created a graphical user interface for controlling and observing the simulations of models through an on-screen mock-up of a mobile phone. Then, we discuss the conrete results we have achieved by using our approach.

1 Introduction

The context of this work is the development of the user interface (UI) software for Nokia's mobile phones. In this domain, the term *feature* means functionality of the phone that is accessible or visible to the user via the UI of the phone. The features are implemented as UI applications in a proprietary mobile phone software architecture. *Feature interaction* means a functional or behavioral (logical) dependency between features. That is features depend on other features to fulfill a service request, or the state of some feature may affect the default behavior or availability of another feature. Interactions between features are necessary and unavoidable, but they can be difficult to control intellectually in the design phase of the UI (conceptual design) and in the implementation and testing of the UI software.

*Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK, email: louisel@daimi.au.dk

†Software Technology Laboratory, Nokia Research Center, P.O. Box 407, FIN-00045 NOKIA GROUP, FINLAND, email:{Antti-Pekka.Tuovinen,Jianli.Xu}@nokia.com

The development of the user interface software for mobile phones is a concurrent and a distributed engineering process. There is a large family of products that share many common features but have also a high degree of variation in terms of supported features and variation in the UI style (size of display, the number and purpose of buttons and keys). There is also strong pressure for (re)using the same SW components in as many products as possible. To avoid costly delays in the integration phase of a set of independently developed features, it is important to identify and clearly specify the feature interactions at an early stage of the development. Precise descriptions of the interactions are also needed when planning the testing of the UI software.

The problem that originally motivated this research work was that feature interactions were not specified explicitly enough in the UI design specifications and the SW design documentation. This meant that SW designers and UI testers had to go through a lot of documentation to find out which interactions should be implemented and tested. Also, not having a clear understanding of the interactions of a new feature caused problems for managers because feature interactions were considered as one of the primary indicators of the cost of developing the new feature. So, having a good understanding of the interactions of the new feature would help managers to make reliable estimates of the cost (in terms of both time and money) of developing the new feature. Therefore, the goals of this work were:

- identifying categories of interactions that are specific to the domain of mobile phones,
- creating behavioral models that capture the typical feature interaction patterns in each category,
- providing an environment for interactive exploration and simulation of the interaction models, and
- providing input to the development of the UI design specification template used by UI designers to document the UIs of features.

In the core of our approach is an executable behavioral model of the underlying UI architecture and the individual features. As the modeling language we use Coloured Petri Nets. Coloured Petri Nets (CP-nets or CPN)[5] is a graphical modelling language with well-defined semantics that allows simulation of CPN models as well as formal analysis [6]. In contrast to many other modelling languages, CP-nets are both state and action oriented. CP-nets are suitable for modelling concurrent systems and a number of projects have demonstrated their usefulness in modelling and analysis of complex systems [10, 8, 3, 4]. The tool Design/CPN [7] that we use makes it possible to add domain specific graphics for visualisation and interaction purposes.

Note that we did not include automatic detection of feature interactions by analytical means (e.g. by state space analysis) as an explicit goal of this research. We first wanted to concentrate on using visualisations and interactive simulation as the means for human users to observe the behavior of the mobile phone when

features interact. Automatical feature interaction detection methods could then be investigated as follow-up work.

The structure of the paper is as follows. In Section 2 we discuss the types of interactions that are typical in mobile phones. Then, in Section 3 we describe at a general level the modelling approach and the use of domain specific graphics in the interactive simulation of the model. Then, in Section 4, we describe the concrete results we have achieved by using our approach and discuss how the results have been used by Nokia Mobile Phones. The technical details of the model and the generation of the visualisations are given in [9].

2 Types of Feature Interactions

The user-interface (UI) of a mobile phone can be characterized as *task-oriented*. This means that the phone UI is designed to directly support the main functions of the device. This is different from a PC that has a generic UI that supports a much wider range of applications. For example, a phone has keys assigned permanently for answering and ending a call. Furthermore, when browsing the contact information stored in the phonebook, it must be possible to call a person by a single press of a key. This design philosophy reflects the requirements and needs of mobile users, and the physical and economical constraints of the devices.

Each mobile phone product follows a certain UI style ranging from basic (simple) models to more complex business-use oriented models and exclusive fashion products. The style is an important part of the product brand and it has a relatively long lifetime. It describes the physical structure of the UI and the basic mechanisms (keys and their operation) of user interaction. The UI specification of a product defines the features of the product by showing the GUI design and by describing the detailed user interaction for each feature.

A mobile phone is a multi-tasking device: many applications (features) can be active at the same time. Even during a phone call and while talking, the user can activate the calendar or start a game through the in-call menu of the phone. The status of external actors can affect how some features behave in a mobile phone. For example, network status affects some features immediately. Also, the accessories that are connected to the phone change the default behavior of some features. From the UI software point of view, these external actors are independent sources of events; they can issue events that the features have to react to and change their behavior accordingly. Inside the phone, the energy manager (battery status supervisor) can also issue events indicating a critical condition (e.g. 'battery low' -event) that need immediate reaction from some applications (a game is stopped) or the whole phone software system (complete shutdown).

It is difficult to manage the behavioral complexity of feature interactions without explicit models of the interactions. When new types of features and more complex UI styles are introduced in the future, the problems become even more difficult. UI designers, UI testers, and UI software developers all need a solid understanding about the implications that feature interactions have on their work.

2.1 Feature Interaction Categories

Feature interactions come from different sources in a mobile phone. The first category of interactions is the use interaction between features. For instance, the task-oriented user interface design requires that when browsing the phone numbers stored in the phone, a call can be made to a number directly from the browser. This represents an interaction between the 'phonebook' and 'mobile originated call' features.

The second category of interactions comes from the need to share the limited UI resources (screen, keypad) between many features that can be activated independent of each other. Because of the prioritization of the features, important events may interrupt less important activities. For example:

- an incoming call screens phonebook browsing for the duration of the call but the browser application does not know it,
- hang-up key stops search from phonebook (the browser is killed), and
- an incoming call suspends a game, but the game is saved and it can be continued.

The third category involves interactions where one feature affects other features by making them unavailable or by modifying their default behavior. For instance, the 'any key answer' feature makes it possible to answer an incoming call by pressing any key on the keypad and the 'key guard' feature locks the keypad for accidental key presses. The combined effect of these features is that if 'any key answer' is enabled and 'key guard' is on, an incoming call can be answered only by pressing the 'send' (off-hook) key. However, once the call is active, 'key guard' is temporarily disabled during the call and then automatically enabled again after the call. This scenario can be made ever more complex by adding other simultaneous events, for instance 'calendar alarm'.

The use interactions are specified in the UI specifications implicitly but in detail. The use of word 'implicit' means that the use interactions are not called interactions directly; they are described as a part of the flow of user's actions. Use interactions are not so difficult to control during implementation. However, the interactions of the second and the third categories are much more difficult to manage and the behavioral specifications of the interactions are scattered in the specifications of the involved features. Therefore, the focus of our work is on modeling and documenting the typical feature interaction patterns that belong to the latter two categories.

3 Modelling and Visualising Feature Interactions

We do not intend to model the entire phone UI software system. Instead, we concentrate on a representative set of features that have interesting interactions. Also, the main focus of the work described here is on the visualisation of feature interactions rather than on the automatic analysis and detection of feature interactions.

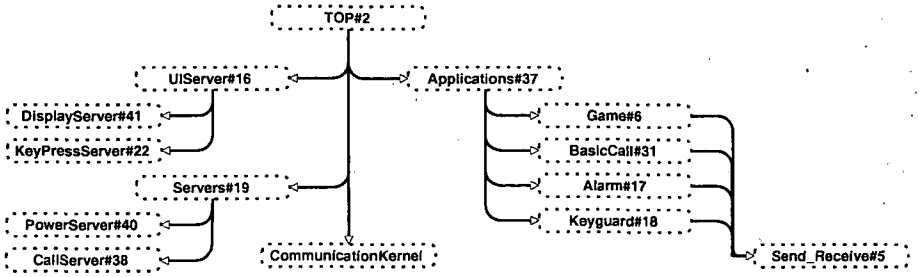


Figure 1: The hierarchy page.

In this section, we first provide an overview of the CPN model. CP-nets will be informally introduced along with the presentation of the model. Then, we explain how we have added domain specific graphics to the model for visualisation and user interaction purposes.

3.1 CPN Models

Figure 1 gives an overview of the CPN model by showing how it has been hierarchically structured into 14 modules (also referred to as subnets or pages). Each node in Fig.1 represents a subnet of the CPN model. An arc between two nodes indicates that the source node contains a substitution transition whose behaviour is described in the subnet represented by the destination node.

The CPN model consists of four main parts that correspond to the four concepts of the phone UI software system: applications, servers, UI controller, and communication kernel. Servers implement the basic capabilities of the phone and Applications implement the behaviour of features by using the services of servers. Applications make the feature available to the user via a user interface; servers do not have user interfaces. Servers and applications communicate by means of asynchronous message passing through the communication kernel. The UI controller manages the user interaction and handles the displays of applications.

The subnet Top depicted in Fig.2 is the top-most page of the CPN model and provides the most abstract view of the CPN model. The page consists of four substitution transitions that correspond to the four parts mentioned above. The detailed behaviour of UIController, Servers, CommunicationKernel, and Applications is modelled in subnets associated with the substitution transitions.

3.1.1 Semantics of CP-nets

A CP-net is created as a graphical drawing with textual inscriptions. A state of a CP-net is represented by means of places which are drawn as ellipses with a name positioned inside. The places contain tokens, which carry data values, or *colours*. Each place has a type (a colour set) which specifies the kind of tokens that the place can hold.

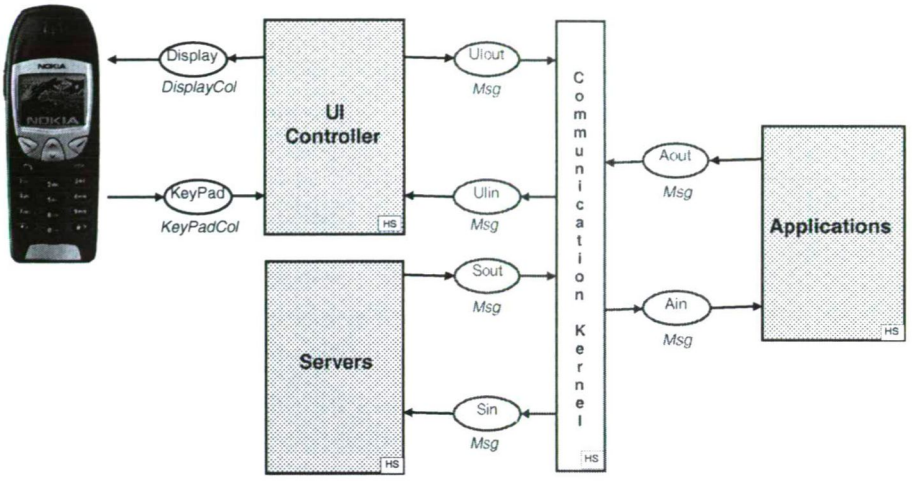


Figure 2: Page Top.

Actions of CP-nets are represented as transitions which are drawn as rectangles with a name positioned inside. The transitions and places are connected by arcs. Transitions remove tokens from places connected by incoming arcs and add tokens to the places connected by outgoing arcs. The tokens removed and added are determined by arc expressions which are textual inscriptions positioned next to the arcs. In the Design/CPN tool, the inscription language is Standard ML. The transitions can have guards that are boolean expressions written in ML.

When simulating a model, a step in a simulation means calculating a new marking for the CP-net. Calculating a new marking means calculating a new assignment of tokens to places as dictated by the firing of a multiset of enabled transitions. Before the transistions can fire, the set of binding elements is calculated first. This means finding all the possible assignments of the tokens in the places to variables (place holders for tokens) in the arc expressions that satisfy the guards of the transitions. Then, a non-conflicting set of binding elements can be chosen to enable a set transitions. Note that a transition can actually be fired more than once during a step depending on the binding elements chosen (see [5] for a detailed definition).

The Design/CPN tool provides ways to control the selection of bindings and the selection of enabled transitions. Furthermore, the tool provides two modes of simulation: interactive and automatic. The interactive mode can be used to debug a CP-net by manually selecting bindings and enabled transitions. In automatic mode, stop criteria can be set to limit the number of steps performed by the simulator.

3.1.2 Extensibility

An important property of the Design/CPN tool is that transitions can have code segments in Standard ML. The code segments are excecuted when transitions fire.

The code segments can employ functions and types imported from user defined ML libraries to create any kind of side effects. For instance, the Mimic library [2] can be used to create animated domain specific graphical representations of the concepts modelled by the CP-nets. Then, the graphical representation can be updated during simulation from the code segments in the transitions to reflect the changes in the state of the model.

The Mimic library provides functions for querying input from the user. That is, the user can point and click the graphical objects on the Mimic windows (called pages). The library captures the raw window events and converts them to ML objects. However, the CP-nets comprising the model must be structured so that they query for user input at the appropriate steps of the simulation. The (blocking) calls to the Mimic query functions are contained in the code segments of transitions.

3.1.3 Modularity

The use of substitution transitions allows the user to relate a transition to a more complex CP-net. The idea is similar to the module concepts found in many programming languages. Substitution transitions can act like macros or functions in terms of how the replacement/substitution is done.

Currently, the model constitutes of 98 page instances. However, there are only 25 different kinds of pages. The applications are modelled using a two or three level hierarchy of pages containing substitution transitions. The state changes of applications are modelled using a generic page that describes a single state transitions in the behavioral model of application. This generic page is hooked up with the input and output message buffers that an application uses to send and receive messages from the UI controller.

Furthermore, CP-nets have the concept of fusion place. This means that the user can specify that a set of places in a CP-net actually represent the same place (object) even though the places of the set are drawn as individual places. So, fusion places have the effect of global variables. Using these two constructs together with Design/CPN's ability to import and export subnets, we have constructed a CPN model of the phone UI software system where features can easily be added and removed. Hence, large parts of the CPN model can be reused in the models of other products with new features.

3.2 Visualisation

An important aspect of our work is the addition of domain specific graphics to the model. This makes it possible to observe, configure and control simulations without interacting directly with the CP-nets. We have made two extensions to the CPN model. First, the state of the phone (as the user observes it) is visualised by an animation of the phone display and the keypad of the phone.

The animation is generated using the Mimic library [2]. So, during an interactive simulation run, the user can press the keys on the animated phone and the

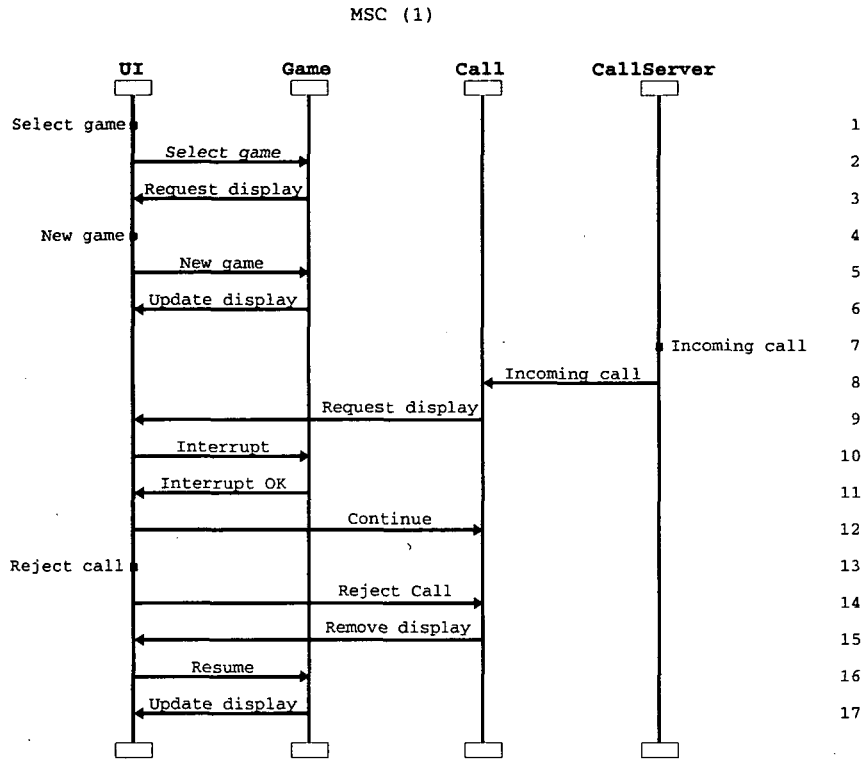


Figure 3: An MSC automatically generated during a simulation run.

underlying model responds correspondingly to the user's actions by changing the display.

Second, the CPN model is extended with Message Sequence Charts (MSCs) [1] that are automatically generated as graphical feedback. The reason MSCs are chosen is that they allow the behaviour of the CPN model to be visualised in a way that is familiar to Nokia's UI designers and software developers.

Fig. 3 shows an example of such an MSC automatically generated during a simulation run of the CPN model. The MSC contains a vertical line for each of the applications and servers in the phone UI software system. The arrows between the vertical lines correspond to messages sent in the system. A mark on a vertical line corresponds to an external event such as an user action. The communication sequence considered corresponds to a scenario where the mobile phone receives an incoming call while the user is playing a game. The scenario demonstrates an interaction between the 'game' and the 'call' features that belongs to the category of UI resource sharing interactions. The scenario consists of the following sequence of events:

- The user selects a game from the menu (line 1)
- The game feature is notified and it requests the display (lines 2-3)
- The user selects a new game (line 4)
- The 'game' feature is notified and it changes the contents of the display accordingly (lines 5-6)
- An incoming call arrives. The 'call' server notifies the 'call' feature (lines 7-8)
- The 'call' feature requests the display (line 9)
- The display is currently in the use of the 'game' feature. The UI controller interrupts the 'game' feature and after the interruption has been acknowledged the display is granted to the 'call' feature (lines 10-12)
- The user rejects the call (line 13)
- The 'call' feature is notified and the display is removed (lines 14-15)
- The 'game' feature is resumed (lines 16-17)

In the scenario above, the UI controller (see Fig. 2) is responsible for interrupting (lines 10-12) and resuming (lines 16-17) the execution of features. So, the features do not have to know which features they potentially interrupt or which features interrupt them. This makes it possible to add and remove features from the CPN model without changing the subnets modelling the rest of the features.

4 Conclusions

4.1 Results

The model provides the basic UI infrastructure where we can plug in features. Currently, the model includes the features listed below. All these features can be simulated by interactive graphical simulation and the corresponding MSCs can be generated during simulation. The MSCs can then be used as static documentation about the events that occurred during simulation and about the messages that were passed in the system during the simulation session.

Idle State
: The Idle State feature handles the mobile phone when it is idle, i.e., is ready to establish incoming and outgoing calls, browse the menus etc.

Menu
: The Menu feature handles the menu structure and browsing of menus of the mobile phone.

Call
: The Call feature handles the incoming and outgoing calls of the mobile phone.

Any Key Answer
: The Any Key Answer allows incoming calls to be answered by the user of the mobile phone by pressing any key of the mobile phone.

Key Guard
: The Key Guard feature blocks the keys of the mobile phone to prevent the user from accidentally pressing keys when not using the phone.

Phone Book
: The Phone Book feature allows names and numbers to be stored in memory.

Profiles and Caller Groups
: The Profiles and Caller Groups feature allows the user to divide the numbers stored in the Phone Book into *caller groups* and assign different ringing tones to each caller group.

Alarm Clock
: The Alarm Clock feature allows the user to set an alarm clock. When the clock expires the mobile phone will indicate the alarm using tones and the mobile phone display.

Power
: The Power feature observes the status of the battery of the mobile phone and indicates when the battery is low or charging.

Game
: The Game feature allows the user to play games.

These features allow us to visualise all three types of interactions described in Section 2. More features were planned to be included in the model (multi-party calls and in-call menu) but these had to be dropped due to the time constraints of the project.

The most valuable result of the modeling work was the understanding of the nature and the causes of feature interactions gained by the people participating in the project. This knowledge was refined and transferred to the customer at the Nokia R&D unit that funded the research.

First, we helped the people responsible for the development of the Nokia mobile phones UI design specifications to add a section about feature interactions into the UI design specification template. The feature interaction section in the new template uses the categorisation described in Section 2. Second, using the MSCs generated from simulating the model, we wrote a document that describes the feature interaction categories with concrete interaction examples. We also identified all the situations caused by events (from internal and external sources) that need to be considered when specifying the UI of a feature.

Naturally, the CPN model and all related technical documentation was also delivered to the customer. The model may be valuable for the future development of the mobile phone UI software architecture. However, from the customer's point of view, the most immediate value of the research comes from the input to the UI specification work.

4.2 Coloured Petri Nets vs. UML models

When we started the work on analysing and modelling feature interactions, we considered also other formalisms than Coloured Petri Nets. The UML statecharts and activity diagrams were also strong candidates. However, we chose CPNs from the following reasons (in no particular order):

- CPNs are both state and action oriented. This gives a choice in the kind of models that are constructed (i.e. data flow or discrete state models).
- The hierarchical structure of nets, the use of separate pages, and the fusion places makes it possible to build modular models using either a top-down or a bottom-up approach.
- Even incomplete models can be immediately simulated (executed) during construction
- Tool support (Design/CPN) for simulation and visualisation of models.
- We had good earlier experiences in using CPNs in modelling and analysing the mobile phone software architecture.

4.3 Final Remarks

The most immediate benefit of our modeling work is the increased understanding of feature interactions. The model provides a sound base to build a body of knowledge about feature interactions in Nokia's mobile phones and how to handle them at the feature level.

We have found that CPNs are well suited for the purpose of modelling feature interactions in mobile phones. As a modelling language, CPNs provides the flexibility and modularity that is needed in the construction of non-trivial models. The extensibility and progammability of CPNs in the Design/CPN tool have been very important properties for our work.

There are several interesting possibilities for future tasks based on the current work. For example, we could link the interaction patterns to existing implementation patterns in the software. This would be achieved through static documentation. Other possible uses of the model include regression testing of the behavior of the phone UI when changing the logic of the features included in the model. The Design/CPN tool gives several possibilities to add support also for automated regression testing. For instance, we could check that invariants expressed as markings (contained tokens) of specific states are preserved (using automatic simulation or state space analysis). In [9], we discuss some initial ideas about automatic detection of feature interactions using the CPN model and some additional tools.

Acknowledgements

This work has been funded by Nokia Mobile Phones. We want to thank Andy Turner, Jyrki Okkonen, Piia Yliranta, and Sirpa Ruokangas for their input.

References

[1] ITU (CCITT). Recommendation Z.120: MSC. Technical report, International Telecommunication Union, 1992.

[2] S. Christensen. *Message Sequence Charts. User's Manual*, January 1997. Available from http://www.daimi.au.dk/designCPN/.

[3] S. Christensen and J.B. Jørgensen. Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In P. Azéma and G. Balbo, editors, *Proceedings of ICATPN'97*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997.

[4] D.J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In J. Billington and W. Reisig, editors, *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 1996.

[5] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[6] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.

[7] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN Reference Manual.* Department of Computer Science, University of Aarhus, Denmark, 1995.
Online: `http://www.daimi.au.dk/designCPN/`.

[8] L. Lorentsen and L.M. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System. In M.Nielsen and D.Simpson, editors, *Proceedings of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 2000.

[9] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling Features and Feature interactions of Nokia Mobile Phones Using Coloured Petri Nets. In J. Esparza and C. Lakos, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN'2002)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[10] J. Xu and J. Kuusela. Analyzing the Execution Architecture of Mobile Phone Software with Colored Petri Nets. *Software Tools for Technology Transfer*, 2(2):133–143, December 1998.

Acta Cybernetica **15** (2002) 633–652.

A Framework for Studying Substitution*

Härmel Nestra†

Abstract

This paper describes a framework for handling bound variable renaming and substitution mathematically rigorously with the aim at the same time to stay as close as possible to human intuitive preconception about the phenomena, so that proofs could be deduced from intuitive motivations more directly than in the case of standard approaches.

The theory is developed for general multi-sorted term algebras with variable binding. Therefore, the results hold for a wide class of term calculi such as the λ-calculus, first-order predicate logic, the abstract syntax of programming languages etc.

1 Introduction

1.1 About the Matter

In the area of formal logic, one can detect some kind of discrepancy between human intuition and the standard treatment of the underlying notions like substitution and bound variable renaming. Namely, these concepts are defined inductively on the structure of a term and therefore also the proofs of theorems concerning them require structural induction. But when we are thinking of substitution or variable renaming intuitively, we imagine direct replacement of subterm occurrences at some places instead. As a consequence, our primary intuitive reasoning of claims concerning substitution and variable renaming turns often out to be useless when a mathematically rigorous proof is needed.

In this paper, we introduce a method for overcoming the discrepancy described. In our algebraic framework, substitution is defined via subterm replacement at arbitrary positions not using induction explicitly. Therefore, the notions of subterm occurrence positions and replacement become underlying in the theory and matters of particular examining. These notions are often used by authors but quite seldom investigated in themselves. Nevertheless, many methods have been developed for handling substitution. In the following, some possible approaches are listed.

- The direct definition by induction on term structure. Bound variable renaming is performed only if this is inevitable. For a typical instance, see [6].

*Partially supported by Estonian Science Foundation under grant no 4155.

†Institute of Computer Science, University of Tartu, J. Liivi 2, 50409 Tartu, Estonia; e-mail: nestra@cs.ut.ee

- The direct definition by induction on term structure assuming the variable convention. The latter simplifies the definition essentially because it excludes the cases where variable capturing arise. For an instance, see [3].
- The direct definition by induction on term structure, but always making bound variable renaming. Despite the increased amount of work in performing substitution, the formal definition is quite short. See [11].
- Defining substitutions as a part of object language, not as a family of meta-level operations. This method is suitable especially for computerizing. See [1] for instance.

1.2 More about This Paper

Substitution is a very general concept in itself, occurring as a matter of investigation whenever one deals with some kind of term system. One aim of this work is to develop all the theory uniformly for a large class of calculi.

In Sect. 2, the concept of term algebras being our setting of reasoning is specified. We give a definition of term algebras which captures the concept of absolutely free first-order multi-sorted algebra. The only formal difference from standard is our use of so-called indices in signature. For treating variable binding, we then add some elements to the signature and refine the term forming rules. Variable binding is encoded into terms standardly using place-holding variables because of our pursuit to stay as close to standard practice as possible.

So we reduce studying variable binding to almost standard universal algebra framework with some restrictions. This is possible when dealing with the formal side of the matter only. Evaluation of the terms cannot go standardly any more, because homomorphisms—the evaluation maps in universal algebra—do not work correctly in the binding case. In this paper, semantics is not considered. For ways to define algebra homomorphisms and its related concepts (subalgebras, quotient algebras etc) in cases where binding is involved, see [12].

Section 3 presents a rigorous treatment of subterm occurrence locations using positions. Our treatment of positions in principle is standard.

In Sect. 4, replacement operators are defined and some of their basic properties are considered. A replacement operator is a mapping from terms to terms whose task is to replace the subterms at some (maybe no) positions simultaneously with some other terms. It is determined by a rule establishing which terms have to be placed at which positions.

There is no need in our treatment for the argument term of a replacement operator to possess all the positions at which replacing should be performed according to the formal rule. The redundant part of the rule is ignored by the operator.

Replacement operators can be defined inductively as in [2]. An example for pure λ-calculus is found also in [6] (Sect. 9A). Inductive definition of replacement is rather troublesome, particularly in the light of the simplicity of the idea behind. Intuitively, if we replace some subterms of a term with some other terms, then in the result term, the new terms occur at positions where the replacing was done

and the rest is as in the original term. We use a definition which is almost direct translation of this intuitive idea to mathematically rigorous form.

In some works on term rewriting, e.g. [10], terms are treated as functions from so-called tree domains to a ranked alphabet. Replacement operation becomes quite elementary in this approach. The price of it is relatively non-traditional, complicated and being far from human intuition treatment of terms. We succeed in maintaining an almost completely standard definition of terms together with easy and intuitive handling of replacement.

The proofs of the theorems about positions and replacement are omitted in this paper (a few theorems are equipped with proof sketches) because replacement is not the main topic of this paper and the claims are mostly intuitively credible.

The purpose of the remaining sections is to exhibit how the theory of variable binding looks in context of our framework. Section 5 gives the definitions, as well as some basic facts. Section 6 develops the theory for α-congruence. Some of the proved facts are common, some are not. Among other things, we express α-congruence as the reflexive transitive closure of a simple relation. In Sect. 7, we define substitution as an operation on α-classes and prove a couple of its nice properties from this aspect.

We have introduced our method earlier in [7, 8] which are preliminary reports of this paper.

1.3 Related Work

Several methods have been developed for describing uniformly all binding situations arising in computer science. Bring up the following.

- The *higher-order abstract syntax* where all possible binding constructs are expressed via λ-terms. This seems to be the most famous and used method of treating arbitrary binding. For example, see [9].
- The *term algebras of binding signature* of [12, 13, 4]. In these papers, also some categorical viewpoints of the area are provided.
- A theory of binding in Fraenkel–Mostowski set theory is presented in [5].
- The *binding structures* of [14]. This work provides a notion of binding algebras which generalize the de Bruijn λ-calculus.

2 Term Algebras

Defining terms in principle requires fixing a signature and some rules for building new terms from given ones. All terms built under the same rules in the same signature form a term algebra with respect to the rules as algebraic operations.

Take the following system for signature.

S1. A set Γ of *types* (or *sorts*).

S2. A set Ω of *term-builders*.

S3. A set $\mathbf{I}_\omega$ of *indices* of ω for every $\omega \in \Omega$. Assume thereby that all the sets $\mathbf{I}_\omega$ are finite and pairwise disjoint. Take $\mathbf{L} = \bigcup_{\omega\in\Omega} \mathbf{I}_\omega$.

S4. A type $\tau_i \in \Gamma$ for each $i \in \mathbf{L}$, and a type $\tau_\omega \in \Gamma$ for each $\omega \in \Omega$.

For each type $\gamma \in \Gamma$, let $\mathcal{X}_\gamma$ be a given set of variables.

T1. For each $\gamma \in \Gamma$, all the variables from the set $\mathcal{X}_\gamma$ are terms of type γ.

T2. Let ω be any term-builder. For every $i \in \mathbf{I}_\omega$, let u_i be any term of type τ_i. Then $\omega(u)$ is a term of type τ_ω, where u denotes the vector which has all the terms u_i as its components, i.e. $u = (u_i : i \in \mathbf{I}_\omega)$.

T3. All terms are constructed by T1 and T2.

T4. Terms constructed in different ways are different.

These rules suggest that, for any $\omega \in \Omega$, the τ_i, $i \in \mathbf{I}_\omega$, could naturally be called the *argument types* of ω and τ_ω the *result type* of ω. Roughly speaking, ω has the function type $(\tau_i : i \in \mathbf{I}_\omega) \to \tau_\omega$.

The indices play the same role here as numbers $1, \ldots, n$ in indexing function arguments usually, $|\mathbf{I}_\omega|$ being the arity of symbol ω. Pairwise disjointness of the index sets is essentially used in proofs of some underlying theorems of the theory, e.g. of Theorem 4.3. Hence using the index sets instead of numbers is substantial.

On the other hand, term construction does not depend on what the elements of $\mathbf{I}_\omega$ actually are, since rule T2 uses u_i in building a new term, not i. Therefore renaming the indices does not change the terms essentially.

Denote our term algebra by $\mathcal{T}$, i.e. $\mathcal{T} = (\mathcal{T}_\gamma : \gamma \in \Gamma)$ where $\mathcal{T}_\gamma$ is the set of all the terms of type γ. The term constructing rules imply that any term which is not a variable has exactly one type. Variables may have several types.

For accommodating variable binding into the framework, a partitioning

$$\mathbf{I}_\omega = \mathbf{A}_\omega \cup \bigcup_{a\in\mathbf{A}_\omega} \mathbf{B}_a \tag{1}$$

is determined for each $\omega \in \Omega$ by the signature (both unions must be disjoint). Call the indices of set $\mathbf{A}_\omega$ the *argument indices* of ω; the others are *binding indices*. Each binding index belongs to exactly one set $\mathbf{B}_a$, being thus associated to a fixed argument index. This is for determining the scope of binding.

The idea is that, in any term of shape $\omega(u)$, the arguments u_b of ω for binding indices b stand for binding occurrences of variables. This is achieved by treating the variables of each type as they were forming a separate type.

To be precise, we have a partitioning $\Gamma = \Gamma \cup \tilde{\Gamma}$ where the elements of Γ are the basic types and the elements of $\tilde{\Gamma} = \{\tilde{\gamma} : \gamma \in \Gamma\}$ are the corresponding variable types. Thereby, τ_b is restricted to be a variable type whenever b is a binding index, while τ_a and τ_ω are restricted to be basic types for every argument index

a and term-builder ω. So each variable has simultaneously its basic type and the corresponding variable type. Assume in the rest for simplicity that each variable has exactly one basic type.

Assume additionally that a term $\omega(u)$, $u = (u_i : i \in \mathbf{I}_\omega)$, can be constructed only if $u_{b_1} = u_{b_2}$ implies $b_1 = b_2$ for any $a \in \mathbf{A}_\omega$ and $b_1, b_2 \in \mathbf{B}_a$. That means, it is allowed to bind one variable only once with the same scope.

For a simple characterizing example, take pure λ-calculus. It has two term-builders—juxtaposition denoting application and the binding symbol λ. As there are no restrictions imposed on the construction of terms except that the first argument of λ must be a variable, pure λ-calculus has one basic type *TERM* and its subtype *VARIABLE*. One may take index sets as in [6] (Sect. 9A): $\mathbf{I}_\cdot = \{1, 2\}$ ($\cdot$ denoting juxtaposition) and $\mathbf{I}_\lambda = \{*, 0\}$, whereby $*$ is the only binding index. So $\tau_1 = \tau_2 = \tau_0 = \mathit{TERM}$ and $\tau_* = \mathit{VARIABLE}$.

For a concrete example, take the λ-term $M = (\lambda x.y)x$ (x and y being variables). Use index 1 for the left-hand argument of juxtaposition and 2 for the other. Then we get $M = \cdot(u)$ where $u = (u_1, u_2)$ with $u_1 = \lambda x.y$ and $u_2 = x$. Further, $u_1 = \lambda(v)$ where $v = (v_0, v_*)$ with $v_0 = y$ and $v_* = x$.

For a bit more complicated example, take untyped first-order predicate logic in some signature. There are two binders $\forall$ and $\exists$. For term-builders, take all the individual, function and predicate symbols of the signature, as well as the propositional connectives. In predicate logic, one distinguishes between so-called "terms" and "formulas". Hence take *TERM* and *FORMULA* to be the two basic types. A variable type *VARIABLE* is corresponding to *TERM* (no propositional variables are used in predicate logic usually, so one can manage without a variable type corresponding to *FORMULA*). One can take the argument index sets arbitrarily in a way that the condition S3 from Sect. 2 holds and the arities match (individual symbols must be taken as 0-ary function symbols). For all function symbols, the argument types equal to *TERM* and also the result type; for all predicate symbols, the argument types equal to *TERM*, but the result type is *FORMULA*. For propositional connectives, the argument types, as well as the result type, equal to *FORMULA*. For quantifiers, the type of the binding argument is *VARIABLE* and the type of the other argument, as well as the result type, is *FORMULA*.

3 Positions and Occurrences

Think of $\mathbf{L}$ as a (possibly infinite) alphabet (recall from Sect. 2 that $\mathbf{L} = \bigcup_{\omega \in \mathbf{\Omega}} \mathbf{I}_\omega$).

Definition 3.1. *Positions* are strings over the alphabet $\mathbf{L}$, i.e. elements of $\mathbf{L}^*$.

Definition 3.2. Define the notion *term t occurs in term s at position p* inductively as follows:

- t occurs in s at ϵ (the empty string) iff $s = t$;
- if $p = p'i$ with $p' \in \mathbf{L}^*$, $i \in \mathbf{L}$, then t occurs in s at p iff a term of shape $\omega(u)$ occurs in s at p', such that $i \in \mathbf{I}_\omega$ and $u_i = t$.

Say that t is a *subterm* of s iff t occurs in s at some position. For intuitive understanding of positions, consider term trees. For each $i \in \mathbf{I}_\omega$, label the edge connecting the root of the tree of $\omega(u)$ with the root of the tree of u_i by i. The position where a subterm occurs in a term is found by writing down in order the labels of edges appearing on the path which connects the root of the term with the root of the subterm. So the only difference from standard treatment of positions is that we use the indices given by signature instead of integers for constructing them.

According to Definition 3.2, only terms of type τ_i can occur in term s at position $p'i$. Therefore define $\tau_{p'i} = \tau_i$ for any $p' \in \mathbf{L}^*$ and $i \in \mathbf{L}$.

Definition 3.2 implies also that, in any term, at most one term can occur at a given position. Denote by $s.p$ the term occurring in s at p, if any. For each $p \in \mathbf{L}^*$, $(.p)$ is a partial function working on terms. However, we treat it as a total function with a special value $\bot$ for denoting undefinedness. So an equality like $s.p = t.p$ is valid if neither $s.p$ nor $t.p$ exists, but if exactly one of $s.p$ and $t.p$ exists, then it is not valid. Assume additionally $\bot.p = \bot$ for every p. Then $s.\epsilon = s$ for all terms s and $s.pq = s.p.q$ for all terms s and positions p, q.

Say that p *is a position of* s iff something occurs in term s at position p, i.e. $s.p \neq \bot$. For all terms s, denote $\operatorname{pos} s = \{p : s.p \neq \bot\}$, i.e. $\operatorname{pos} s$ is the set of all positions of s. Clearly $\operatorname{pos} x = \{\epsilon\}$ for any variable x.

Definition 3.3. (i) Let $p, q \in \mathbf{L}^*$. Say that q is a *refinement* of p iff p is a prefix of q, i.e. $q = pr$ for some $r \in \mathbf{L}^*$. If this holds, we write $p \leq q$ (or $q \geq p$).

(ii) If $p, q, r \in \mathbf{L}^*$ with $q = pr$, then write $q\big|_p$ for r.

(iii) For arbitrary $Q \subseteq \mathbf{L}^*$ and $p \in \mathbf{L}^*$, define $pQ = \{pq : q \in Q\}$ and $Q\big|_p = \{r : pr \in Q\} = \left\{q\big|_p : q \in Q, p \leq q\right\}$.

(iv) Let $p, q \in \mathbf{L}^*$. If neither p is a prefix of q nor q is a prefix of p, then p and q are called *divergent* and denoted $p \asymp q$.

(v) Let $P \subseteq \mathbf{L}^*$ and $q \in \mathbf{L}^*$. If $q \asymp p$ for each $p \in P$, then call q *divergent from* P and write $q \asymp P$.

For a concrete example, take the λ-term $M = (\lambda x.y)x$ studied in Sect. 2. Choose the argument indices as above. Then the term $(\lambda x.y)$ occurs in M at position 1 and the term x occurs in M at position 2. Analogously, y occurs in $\lambda x.y$ at position 0 and x occurs in $\lambda x.y$ at position $*$. So the term y occurs in M at position 10 and the term x occurs in M also at position $1*$. Moreover, M itself occurs in M at position ϵ. As the variables are not constructed terms but primitive, no terms occur in variables at non-empty positions. Thus we can find no other occurrences of terms in M, so $\operatorname{pos} M = \{\epsilon, 1, 2, 10, 1*\}$. Among the positions of M, 1 and 2 are divergent, 10 and $1*$ are divergent, and 2 is divergent from both 10 and $1*$.

The relation $\leq$ is a partial order on $\mathbf{L}^*$. So one can speak about maximality and minimality with respect to the relation $\leq$. If $P \subseteq \mathbf{L}^*$, then any two positions both maximal in P are divergent. The same holds for positions minimal in P. If p and q are divergent, then any refinements of these are, too. Any set $Q \subseteq \mathbf{L}^*$

whose elements are pairwise divergent is called *antichain*, whereby Q is said to be an antichain of P whenever $Q \subseteq P$.

The following is one of the theorems underlying our theory. The proof uses essentially the pairwise disjointness of the index sets. Note that we write Σ^r, Σ^t and Σ^c for reflexive, transitive and compatible, respectively, closure of relation Σ. Recall from universal algebra that a binary relation Σ is called compatible iff, for any $\omega \in \mathbf{\Omega}$ and vectors $(u_i : i \in \mathbf{I}_\omega)$ and $(v_i : i \in \mathbf{I}_\omega)$, $(u_i, v_i) \in \Sigma$ for all $i \in \mathbf{I}_\omega$ implies $(\omega(u), \omega(v)) \in \Sigma$.

Theorem 3.4. *Let Σ be a binary relation on $\mathcal{T}$ and s, t be terms. Then $(s,t) \in \Sigma^c$ iff there exists a common maximal w.r.t. set inclusion antichain P of both* pos s *and* pos t *such that $(s\,.\,p, t\,.\,p) \in \Sigma$ for all $p \in P$.*

Proof. ($\Rightarrow$) Build the compatible closure of Σ iteratively. Argue by induction on the number of steps it takes to get (s,t).

($\Leftarrow$) Let $\|P\|$ be the sum of the lengths of positions of P. The claim follows by induction on $\|P\|$. □

As a corollary, we get the following theorem which provides a method for proving that terms are congruent if we know that some corresponding subterms of them are congruent. Note that equality is just a particular congruence relation.

Theorem 3.5. *Let $\equiv$ be any congruence relation on the term algebra $\mathcal{T}$. Let s, t be terms and $P \subseteq \operatorname{pos} s \cap \operatorname{pos} t$ an antichain. If $s\,.\,p \equiv t\,.\,p$ for each $p \in P$ and $s\,.\,q = t\,.\,q$ for each $q \prec P$, then $s \equiv t$.*

4 Replacement Operators

Definition 4.1. Call a function f a *placing rule* iff its domain dom f is an antichain of $\mathbf{L}^*$ and $f(p) \in \mathcal{T}_{\tau_p}$ for every non-empty $p \in \operatorname{dom} f$.

Definition 4.2. If f is a function with domain P and $q \in \mathbf{L}^*$, then $f\big|_q$ denotes the function with domain $P\big|_q$ and $f\big|_q(r) = f(qr)$ for each $r \in P\big|_q$.

Note that $f\big|_q$ is a placing rule whenever f is.

Our treatment of replacement is grounded on the following theorem.

Theorem 4.3. *Let s be a term and f be a placing rule with domain $P \subseteq \operatorname{pos} s$. Then there exists a unique term t such that $t.p = f(p)$ for each $p \in P$ and $t.q = s.q$ for each $q \prec P$. Thereby, if $\epsilon \notin P$ then t is of the same type as s.*

Proof. Argue by induction on $\|P\|$ as in Theorem 3.4. An alternative way is to prove the claim for singleton sets P at first, using induction on the only element of P, and generalize to any P by induction on $|P|$. The uniqueness part of this theorem can be deduced also from Theorem 3.5. □

Theorem 4.3 justifies the following definition.

Definition 4.4. Let s be a term. Let f be a placing rule.

(i) If $\operatorname{dom} f = P \subseteq \operatorname{pos} s$, then define $[f](s)$, the result of simultaneous replacement of the subterms of s at positions $p \in P$ by corresponding terms $f(p)$, to be the unique term whose existence is claimed by Theorem 4.3.

(ii) If $\operatorname{dom} f \not\subseteq \operatorname{pos} s$, then define $[f](s) = [f|_{\operatorname{pos} s}](s)$ where $f|_{\operatorname{pos} s}$ is the function with domain $\operatorname{dom} f \cap \operatorname{pos} s$ behaving like f on it.

Hence replacement operators are defined for cases only for which the positions where the replacement must be performed simultaneously are pairwise divergent.

The replacement operator $[f]$ will frequently be denoted similarly to set comprehension syntax by $[f(p) : p \in \operatorname{dom} f]$ (with some concrete expressions at place of $f(p)$ and $\operatorname{dom} f$ of course; we often practise even writings like $[z : r \in R]$— this means that the placing rule is constantly z with domain R). Assuming $\operatorname{dom} f = \{p_1, \ldots, p_n\}$ and $u_i = f(p_i)$ for each $i = 1, \ldots, n$, one can write also $[p_1 \mapsto u_1, \ldots, p_n \mapsto u_n]$ instead of $[f]$.

Theorem 4.5. *Let s be a term. Let f be a placing rule. Denoting $P = \operatorname{dom} f \cap \operatorname{pos} s$, we have*

$$\operatorname{pos}[f](s) = \bigcup_{p \in P} (p \operatorname{pos} f(p) \cup \{r : r < p\}) \cup \{q \in \operatorname{pos} s : q \asymp P\}.$$

If we replace variables with variables only, then P is a subset of all positions maximal in $\operatorname{pos} s$ and, for each $p \in P$, $\operatorname{pos} f(p) = \{\epsilon\}$. Hence Theorem 4.5 implies that replacing variables with variables does not change the set of positions.

The following theorems state some basic properties of replacement. Theorem 4.7 is for computing expressions of form $[f](s) \cdot q$. Theorem 4.8 states that if the replacement operators do not "disturb" each other, then the order of their application is unimportant. Theorem 4.9 states some conditions under which one replacement operator absorbs another in consecutive application. Note that if function composition is denoted by ;, then the left function is applied first.

Theorem 4.6. *Let s be a term and f a placing rule such that $f(p) = s \cdot p$ for each $p \in \operatorname{dom} f \cap \operatorname{pos} s$. Then $[f](s) = s$.*

Theorem 4.7. *Let s be a term, f be a placing rule and $q \in \mathbf{L}^*$.*

(i) *If $p \leq q$ for some $p \in \operatorname{dom} f \cap \operatorname{pos} s$, then $[f](s) \cdot q = f(p) \cdot q|_p$.*

(ii) *If $p < q$ for no $p \in \operatorname{dom} f \cap \operatorname{pos} s$, then $[f](s) \cdot q = [f|_q](s \cdot q)$.*

Note that the conditions of Theorem 4.7 (ii) hold whenever $q \leq p$ for some $p \in \operatorname{dom} f \cap \operatorname{pos} s$ because $\operatorname{dom} f$ is an antichain.

Theorem 4.8. *Let $f_1, \ldots, f_n$ be placing rules. Assume that the positions of $\operatorname{dom} f_i$ are divergent from the positions of $\operatorname{dom} f_j$ whenever $i \neq j$. Then $[f_1]; \ldots; [f_n] = [h]$ where $\operatorname{dom} h = \operatorname{dom} f_1 \cup \ldots \cup \operatorname{dom} f_n$ and $h(p) = f_i(p)$ whenever $p \in \operatorname{dom} f_i$. So the composition does not depend on the order of application.*

Theorem 4.9. *Let f, g be placing rules such that each element of* $\operatorname{dom} f$ *has a prefix in* $\operatorname{dom} g$. *Then* $[f]\,;[g] = [g]$.

Theorems 4.9, 4.7 (ii) and 4.6 together give the following.

Theorem 4.10. *Let s be a term and f a placing rule. Let Q be an antichain of* $\operatorname{pos} s$ *such that every $p \in \operatorname{dom} f \cap \operatorname{pos} s$ has a prefix in Q. Then* $[f](s) = [[f|_q](s.q) : q \in Q](s)$.

These properties of replacement are intuitively rather clear, therefore we will often use them in the rest without explicit mentioning.

We now discuss briefly the so-called replacement property. This is one of the main properties assumed about rewrite relations in term rewriting theory.

Definition 4.11. A binary relation Σ on algebra $\mathcal{T}$ is said to have the *replacement property* iff, for any terms s, u and position $p \in \operatorname{pos} s$, $(s \,.\, p, u) \in \Sigma$ implies $(s, [p \mapsto u](s)) \in \Sigma$.

In other words, a relation Σ has the replacement property iff replacing any subterm of s with a term related to it gives a term related to s.

The following theorem shows the connections between the replacement property and compatibility.

Theorem 4.12. (i) *Any reflexive compatible relation Σ on $\mathcal{T}$ has the replacement property.*

(ii) *Any transitive relation having the replacement property is compatible.*

(iii) *The reflexive closure of any relation having the replacement property has the replacement property.*

(iv) *The transitive closure of any relation having the replacement property both is compatible and has the replacement property.*

5 Variable Binding in Terms

Our treatment of binding uses positions essentially: binding of positions is the primary, binding of variable occurrence the secondary notion.

Definition 5.1. Call a position q *binding* iff $q = pb$ for some $p \in \mathbf{L}^*$, $b \in \mathbf{B}_a$, $a \in \mathbf{A}_\omega$ and $\omega \in \mathbf{\Omega}$. Thereby call position pa the *root of binding* corresponding to pb.

If either a binding position pb or its root of binding pa, where $b \in \mathbf{B}_a$ and $a \in \mathbf{A}_\omega$, belongs to $\operatorname{pos} s$, then $s \,.\, p = \omega(u)$ for some vector $u = (u_i : i \in \mathbf{I}_\omega)$ which gives $s \,.\, pb = u_b$ and $s \,.\, pa = u_a$. Hence a binding position belongs to $\operatorname{pos} s$ iff its root of binding does.

Definition 5.2. Let x be a variable and s a term.

(i) Let p be a position. If $s \,.\, pb = x$ for $b \in \mathbf{B}_a$, $a \in \mathbf{A}_\omega$ and $\omega \in \mathbf{\Omega}$, then say that the occurrence of x at pb in s is *binding*, or equivalently, x *occurs binding* at pb in s. We may say additionally that position pa is the root of binding of x.

(ii) Let p, r be positions of s. Take $\omega \in \Omega$, $a \in \mathbf{A}_\omega$ and $b \in \mathbf{B}_a$ such that x occurs binding at pb in s. We say that the occurrence of x at pb in s *binds* position r iff r is not a binding position and pa is the longest prefix of r being a root of binding of x in s. If additionally $s \cdot r = x$, then say that the occurrence of x at pb binds the occurrence of x at r.

(iii) Let r be a position of s. Say that x is *bound* at r in s iff there exists an occurrence of x in s which binds it at r. If additionally $s \cdot r = x$, then say that the occurrence of x at r in s is bound, or equivalently, x *occurs bound* at r in s.

(iv) Let r be a position of s. Say that x is *free* at r in s iff r is not binding and x is not bound in r. If additionally $s \cdot r = x$, then say that the occurrence of x at r in s is free, or equivalently, x *occurs free* at r in s.

Note that an occurrence of x can be binding even if there are no occurrences of x bound by that occurrence. Moreover, note that x can be bound or free at r even if it does not occur at r.

In the paper [7] of ours, binding was defined in a slightly simpler way. This was possible because of some more restrictions imposed on terms there.

Proposition 5.3. *Let s be a term, x a variable and r a non-binding position of s.*
(i) *x is bound at r in s iff some prefix of r is a root of binding of x in s.*
(ii) *x is free at r in s iff no prefix of r is a root of binding of x in s.*

Proof. Straightforward by Definition 5.2. □

Definition 5.4. (i) Let $q \in \operatorname{pos} s$ be any binding position and $s \cdot q = x$. The set consisting of position q, as well as all positions r such that $s \cdot r = x$ and the occurrence of x at q binds the occurrence of x at r, is called the *binding unit* (of x) corresponding to q in s and denoted by $\operatorname{bu}(q, s)$.

(ii) Let s be a term. For any variable x, let $\operatorname{fpos}_x s$ denote the set of all positions where x occurs free in s, and $\operatorname{bpos}_x s$ denote the set of all positions where x occurs non-free (i.e. binding or bound) in s. Define

$$\operatorname{fpos} s = \bigcup_x \operatorname{fpos}_x s, \qquad \operatorname{bpos} s = \bigcup_x \operatorname{bpos}_x s. \tag{2}$$

(iii) For term s, let $\operatorname{fv} s$ denote the set of all variables having a free occurrence in s, and $\operatorname{bv} s$ denote the set of all variables having a binding occurrence in s.

Proposition 5.5. *Let s be a term and p its position. Then $\operatorname{fpos}(s.p) = \operatorname{fpos} s\big|_p \cup R$ where R is the set consisting of all positions where some variable not being free at p in s occurs free in $s \cdot p$.*

Proof. If a variable occurs free at pr in s, then it clearly occurs free at r in $s \cdot p$. A variable occurring free at r in $s \cdot p$ but non-free at pr in s means that the root of binding pr at s is a prefix of p, i.e. the variable is not free at p in s. □

Definition 5.6. (i) Let ϱ be any type-preserving mapping of variables to variables. Define the *naive renaming* $\lceil\varrho\rceil$ on arbitrary term s by $\lceil\varrho\rceil(s) = [\varrho(s \cdot p) : p \in \mathrm{bpos}\, s](s)$.

(ii) Let σ be any type-preserving mapping of variables to terms. Define the *naive substitution* $\lfloor\sigma\rfloor$ on arbitrary term s by $\lfloor\sigma\rfloor(s) = [\sigma(s \cdot p) : p \in \mathrm{fpos}\, s](s)$.

(iii) For any mapping σ of variables to terms, define $\mathrm{supp}\,\sigma = \{x : \sigma(x) \neq x\}$ (the *support* of σ). The application of $\lfloor\sigma\rfloor$ to s is called *sound* iff $\mathrm{bv}\, s$ is disjoint with $\mathrm{fv}\,\sigma(x)$ whenever $x \in \mathrm{fv}\, s \cap \mathrm{supp}\,\sigma$.

Naive renaming and substitution are not entirely satisfactory because variable capturing has not been taken into account. They provide a starting point for defining and studying correct capture-avoiding substitution. Correct bound variable renaming is the matter of the next definition.

Definition 5.7. (i) Let s be a term and q a binding position of s. Let z be a variable of type τ_q. Define $\mathrm{ren}_{q\to z}(s) = [z : p \in \mathrm{bu}(q, s)](s)$.

(ii) Say that terms s, t are in relation A (the renaming-step relation) iff $t = \mathrm{ren}_{pb\to z}(s)$ for some binding position pb, $b \in \mathbf{B}_a$, of s and a *fresh* variable z, i.e. z not occurring in $s \cdot pa$.

(iii) The least congruence relation containing A is denoted by α.

Proposition 5.8. *Let s be any term.*

(i) *For any type-preserving mapping ϱ of variables to variables,* $\mathrm{pos}\lceil\varrho\rceil(s) = \mathrm{pos}\, s$.

(ii) *If t is a term such that $(s, t) \in \mathrm{A}$, then* $\mathrm{pos}\, s = \mathrm{pos}\, t$.

Proof. By the remark made after Theorem 4.5. □

The following three lemmas are rather straightforward corollaries of the definitions presented above. Their proofs however require quite technical and uninteresting study of details, so we omit them as the aim of this paper is to exhibit the proofs of more complicated theorems in the framework developed.

Lemma 5.9. *Let s be a term, $r \in \mathrm{pos}\, s$ and q a binding position of $s \cdot r$. Then* $\mathrm{bu}(q, s \cdot r) = \mathrm{bu}(rq, s)\big|_r$.

Lemma 5.10. *Let s be a term and pb, $b \in \mathbf{B}_a$, a binding position of s with $s \cdot pb = x$. Let f be a placing rule such that* $\mathrm{dom}\, f$ *contains neither binding positions nor prefixes of p, and x occurs free in $f(r)$ for no $r \in \mathrm{dom}\, f$. Then* $\mathrm{bu}(pb, [f](s)) = \mathrm{bu}(pb, s) \setminus \bigcup_{r\in\mathrm{dom}\, f} \{q : r \leq q\}$.

Lemma 5.11. *Let s, t be terms such that $(s, t) \in \mathrm{A}$.*

(i) *Terms s and t have same binding units.*

(ii) *For every variable x,* $\mathrm{fpos}_x\, s = \mathrm{fpos}_x\, t$.

We end the section with proving three propositions using the facts stated so far.

Proposition 5.12. *Relation* A *has the replacement property.*

Proof. Take a term s and a position $r \in \operatorname{pos} s$. Assume $(s \cdot r, u) \in \mathrm{A}$, i.e. $u = \operatorname{ren}_{pb \to z}(s \cdot r)$ for some binding position pb, $b \in \mathbf{B}_a$, of $s \cdot r$ and z not occurring in $s \cdot r \cdot pa = s \cdot rpa$. Now

$$\begin{aligned}
[r \mapsto u](s) &= [r \mapsto \operatorname{ren}_{pb \to z}(s \cdot r)](s) \\
&= [r \mapsto [z : q \in \operatorname{bu}(pb, s \cdot r)](s \cdot r)](s) && \text{(by 5.7 (i))} \\
&= [r \mapsto [z : q \in \operatorname{bu}(rpb, s)\big|_r](s \cdot r)](s) && \text{(by 5.9)} \\
&= [z : q \in \operatorname{bu}(rpb, s)](s) && \text{(by 4.10)} \\
&= \operatorname{ren}_{rpb \to z}(s) && \text{(by 5.7 (i))}
\end{aligned}$$

gives the desired result. □

Proposition 5.13. *Let s be a term and q its binding position. Let σ be a type-preserving mapping of variables to terms such that applying $\lfloor\sigma\rfloor$ to s is sound. Then* $\operatorname{bu}(q, \lfloor\sigma\rfloor(s)) = \operatorname{bu}(q, s)$.

Proof. Theorem 4.5 implies $\operatorname{pos} s \subseteq \operatorname{pos} \lfloor\sigma\rfloor(s)$. Thus $\lfloor\sigma\rfloor(s)$ has the same binding positions and the same roots of bindings as s plus maybe some more which do not belong to $\operatorname{pos} s$. For any $r \asymp \operatorname{fpos} s$, among the rest for each $r \in \operatorname{bpos} s$, we have $\lfloor\sigma\rfloor(s) \cdot r = s \cdot r$. Consequently $\operatorname{bu}(q, s) \subseteq \operatorname{bu}(q, \lfloor\sigma\rfloor(s))$ for every binding position $q \in \operatorname{pos} s$, and any position being divergent from $\operatorname{fpos} s$ belongs to $\operatorname{bu}(q, s)$ iff it belongs to $\operatorname{bu}(q, \lfloor\sigma\rfloor(s))$. Consider the case $r \not\asymp \operatorname{fpos} s$ now. Take $p \in \operatorname{fpos} s$ such that $r \not\asymp p$. If $r < p$, then $r \notin \operatorname{bu}(q, s)$, as well as $r \notin \operatorname{bu}(q, \lfloor\sigma\rfloor(s))$, since binding units contain maximal positions only. If $r \geq p$, then $r \notin \operatorname{bu}(q, s)$, but $r \notin \operatorname{bu}(q, \lfloor\sigma\rfloor(s))$ either because otherwise the variable $s \cdot q$ would occur free at $r\big|_p$ in $\sigma(s \cdot p)$ contradicting the soundness. Hence the claim follows. □

Proposition 5.14. *Let s be a term and $p \in \operatorname{pos} s$. Let σ be a type-preserving mapping of variables to terms. Then $\lfloor\sigma\rfloor(s) \cdot p = \lfloor\sigma'\rfloor(s \cdot p)$ where*

$$\sigma'(x) = \begin{cases} \sigma(x), & \text{if } x \in \operatorname{supp}\sigma \text{ and } x \text{ is free at } p \text{ in } s, \\ x & \text{otherwise.} \end{cases}$$

Proof. Proposition 5.3 (ii) implies that a set of positions where a certain variable is free is closed w.r.t. taking prefixes. This gives that $\sigma'(s \cdot pr) = \sigma(s \cdot pr)$ for each $r \in \operatorname{fpos} s\big|_p$. Denote by R the set consisting of all positions where some variable not being free at p in s occurs free in $s \cdot p$. Then $\sigma'(s \cdot pr) = s \cdot pr$ for each $r \in R$. We have

$$\begin{aligned}
\lfloor\sigma\rfloor(s) \cdot p &= [\sigma(s \cdot r) : r \in \operatorname{fpos} s](s) \cdot p && \text{(by 5.6 (ii))} \\
&= [\sigma(s \cdot pr) : r \in \operatorname{fpos} s\big|_p](s \cdot p) && \text{(by 4.7 (ii))} \\
&= [\sigma(s \cdot pr) : r \in \operatorname{fpos} s\big|_p]([s \cdot pr : r \in R](s \cdot p)) && \text{(by 4.6)} \\
&= [\sigma'(s \cdot pr) : r \in \operatorname{fpos} s\big|_p]([\sigma'(s \cdot pr) : r \in R](s \cdot p)) \\
&= [\sigma'(s \cdot pr) : r \in \operatorname{fpos} s\big|_p \cup R](s \cdot p) && \text{(by 4.8)} \\
&= [\sigma'(s \cdot pr) : r \in \operatorname{fpos}(s \cdot p)](s \cdot p) && \text{(by 5.5)} \\
&= \lfloor\sigma'\rfloor(s \cdot p). && \text{(by 5.6 (ii))}
\end{aligned}$$

□

6 Investigating α-congruence

6.1 Expressing α as reflexive transitive closure

Lemma 6.1. *Let s be a term and $p_1b_1, \ldots, p_nb_n$, $n > 0$, different binding positions in s, $b_i \in \mathbf{B}_{a_i}$ for each $i = 1, \ldots, n$. Let ϱ be a type-preserving injective on its support mapping of variables to variables such that $s \cdot p_ib_i \in \operatorname{supp}\varrho$ and $\varrho(s \cdot p_ib_i)$ does not occur in $s \cdot p_ia_i$ for any $i = 1, \ldots, n$. Let f be the placing rule with $\operatorname{dom} f = \bigcup_{i=1}^{n} \operatorname{bu}(p_ib_i, s)$ and $f(r) = \varrho(s.r)$ for all $r \in \operatorname{dom} f$. Then $(s, [f](s)) \in \mathrm{A}^t$.*

Proof. Without loss of generality, assume the ordering $p_1, \ldots, p_n$ being such that, for all $i, j = 1, \ldots, n$, $p_ia_i < p_ja_j$ implies $i < j$. Denote $x_i = s \cdot p_ib_i$ and $z_i = \varrho(x_i)$. Define $s_0 = s$ and $s_{i+1} = \operatorname{ren}_{p_{i+1}b_{i+1} \to z_{i+1}}(s_i)$ for all $i = 0, \ldots, n-1$.

Prove now that $(s_i, s_{i+1}) \in \mathrm{A}$ for all $i = 0, \ldots, n-1$, i.e. z_{i+1} does not occur in $s_i \cdot p_{i+1}a_{i+1}$. Suppose the contrary, i.e. $s_i \cdot p_{i+1}a_{i+1} \cdot r = z_{i+1}$ for some r. Assume thereby that i is the least number for which such situation arises. This ensures that terms $s_0, \ldots, s_i$ have same binding units by Lemma 5.11 (i). Since $s \cdot p_{i+1}a_{i+1}r \neq z_{i+1}$ by conditions, we can find the biggest k such that $s_k \cdot p_{i+1}a_{i+1}r \neq z_{i+1}$. Therefore $p_{i+1}a_{i+1}r$ belongs to the binding unit corresponding to $p_{k+1}b_{k+1}$. This implies $z_{i+1} = z_{k+1}$ giving $x_{i+1} = x_{k+1}$ by the injectivity of ϱ.

If $p_{i+1}a_{i+1}r$ is a binding position, then $p_{i+1}a_{i+1}r = p_{k+1}b_{k+1}$ is the only possibility. Since $p_{i+1}a_{i+1}$ is not binding, it must be $r = p'b_{k+1}$ for some p'. So $p_{i+1}a_{i+1} \leq p_{i+1}a_{i+1}p' = p_{k+1} < p_{k+1}a_{k+1}$ giving $i < k$, a contradiction. If $p_{i+1}a_{i+1}r$ is not binding, then it is bound by the occurrence at $p_{k+1}b_{k+1}$, so $p_{k+1}a_{k+1}$ is the longest prefix of $p_{i+1}a_{i+1}r$ being a root of binding of x_{i+1}. Thus $p_{i+1}a_{i+1} \leq p_{k+1}a_{k+1}$. Strict inequality is impossible as earlier, so $p_{i+1}a_{i+1} = p_{k+1}a_{k+1}$. But this implies b_{i+1} and b_{k+1} both belonging to $\mathbf{B}_a$ for $a = a_{i+1} = a_{k+1}$. By the restriction imposed on term construction rules in Sect. 2, these positions cannot bind the same variable, a contradiction.

It remains to prove $s_n = [f](s)$. We have

$$\begin{aligned} s_{i+1} &= \operatorname{ren}_{p_{i+1}b_{i+1} \to z_{i+1}}(s_i) = \\ &= [z_{i+1} : r \in \operatorname{bu}(p_{i+1}b_{i+1}, s_i)](s_i) = \\ &= [z_{i+1} : r \in \operatorname{bu}(p_{i+1}b_{i+1}, s)](s_i). \end{aligned}$$

So s_n is expressed as an application of a composition of replacement operators to $s_0 = s$. This composition equals to $[f]$ by Theorem 4.8 which applies since positions of different binding units of the same term are divergent. □

Corollary 6.2. *Let s be a term. Let ϱ be a type-preserving injective on its support mapping of variables to variables such that $\varrho(x)$ does not occur in s for any $x \in \operatorname{supp}\varrho$. Then $(s, \lceil\varrho\rceil(s)) \in \mathrm{A}^{rt}$.*

Proof. If the variables of $\operatorname{supp}\varrho$ do not occur binding in s, then $\lceil\varrho\rceil(s) = s$ and we have done. Otherwise let $p_1b_1, \ldots, p_nb_n$ be the binding positions where variables of $\operatorname{supp}\varrho$ occur in s. Define f as in the formulation of Lemma 6.1. Then Lemma 6.1 applies and also the result follows since $\lceil\varrho\rceil(s) = [f](s)$. □

Lemma 6.3. *Let s be a term and pb its binding position, $b \in \mathbf{B}_a$. Let z be a variable having the same type as x and not occurring in $s \cdot pa$. Let f be the placing rule with $\operatorname{dom} f = \{pb\} \cup \{paq : s \cdot paq = x\}$ and $f(r) = z$ for all $r \in \operatorname{dom} f$. Let g be the placing rule with $\operatorname{dom} g = \operatorname{dom} f$ and $g(r) = x$ for all $r \in \operatorname{dom} g$. Denote $u = [f](s)$ and $t = \operatorname{ren}_{pb \to z}(s)$.*

(i) $\operatorname{dom} g = \{pb\} \cup \{paq : u \cdot paq = z\}$ *and* $s = [g](u)$, *whereby x does not occur in $u \cdot pa$.*

(ii) $(s,t) \in \mathrm{A}$, $(t,u) \in \mathrm{A}^t$.

(iii) $(u,s) \in \mathrm{A}^t$.

Proof. (i) We must show that $s \cdot paq = x \iff u \cdot paq = z$. It suffices to consider positions paq maximal in s (hence also in u). If $s \cdot paq = x$, then $paq \in \operatorname{dom} f$, therefore $u \cdot paq = [f](s) \cdot paq = f(paq) = z$. If $s \cdot paq \neq x$, then $paq \notin \operatorname{dom} f$ which implies $paq \asymp \operatorname{dom} f$ because paq and the positions of $\operatorname{dom} f$ are all maximal. Thus $u \cdot paq = [f](s) \cdot paq = s \cdot paq \notin \{x, z\}$. This proves also that x does not occur in $u \cdot pa$. Now $[g](u) = [g]([f](s)) = [g](s) = s$ by Theorems 4.9 and 4.6.

(ii) Any occurrence of x at a position of $\operatorname{dom} f$ neither is free nor is bound by an occurrence at a position outside $\operatorname{dom} f$. Hence $\operatorname{dom} f$ partitions into binding units of x, whereby all the roots of binding of them are refinements of pa, so pa is the least among them. Since z does not occur in $s \cdot pa$, it does not occur in $s \cdot par$ for any root of binding par. Defining ϱ with $\operatorname{supp} \varrho = \{x\}$, $\varrho(x) = z$, Lemma 6.1 gives $(s,u) \in \mathrm{A}^t$, whereby the proof of it gives more precisely $(s,t) \in \mathrm{A}$ and $(t,u) \in \mathrm{A}^t$.

(iii) Part (i) of this lemma proves the assumptions of this lemma for the case of taking u, s, z and g at place of s, u, x and f, respectively. Part (ii) gives then $(u,s) \in \mathrm{A}^t$. □

Claim 6.4. *Let s,t be terms. If $(s,t) \in \mathrm{A}$, then $(t,s) \in \mathrm{A}^t$.*

Proof. If $(s,t) \in \mathrm{A}$, then $t = \operatorname{ren}_{qb \to z}(s)$ for some binding position qb, $b \in \mathbf{B}_a$, of s and variable z not occurring in $s \cdot pa$. Denote $x = s \cdot pb$ and let f be the placing rule with $\operatorname{dom} f = \{pb\} \cup \{paq : s \cdot paq = x\}$ and $f(r) = z$ for all $r \in \operatorname{dom} f$. If we take $u = [f](s)$, then Lemma 6.3 gives $(t,u) \in \mathrm{A}^t$ and $(u,s) \in \mathrm{A}^t$. Hence $(t,s) \in \mathrm{A}^t$ by transitivity. □

Theorem 6.5. $\alpha = \mathrm{A}^{rt}$.

Proof. We have $\mathrm{A}^{rt} \subseteq \alpha$ by definition of α. For the opposite inclusion, we must show that A^{rt} is a congruence. As it is reflexive and transitive, it suffices to show symmetry and compatibility. Compatibility follows from Theorems 5.12 and 4.12 (iii), (iv). For symmetry, apply Claim 6.4 iteratively. □

Using this theorem, we can simply generalize some facts about relation A to α-congruence as in the following corollary.

Corollary 6.6. *Let s,t be α-congruent terms. Then:*

(i) $\operatorname{pos} s = \operatorname{pos} t$;

(ii) *s and t have same binding units;*

(iii) *for any variable x,* $\operatorname{fpos}_x s = \operatorname{fpos}_x t$;

(iv) $\operatorname{fv} s = \operatorname{fv} t$.

Corollary 6.6 can be proved also without using the theory of this subsection. For (i), construct an algebra in our signature such that pos appears to be a homomorphism from $\mathcal{T}$ to it. Then the kernel relation of pos is a congruence containing A. Now use the definition of α. The other equalities can be proved analogously.

Theorem 6.5 helps also to prove the following fact.

Claim 6.7. *Let s, t be α-congruent terms. Then $s.q = t.q$ for all binding positions q implies $s = t$.*

Proof. Let P be the set of positions maximal in s (so also in t), and take $p \in P$. If $p \in \mathrm{bu}(q, s)$ for some q (so also $p \in \mathrm{bu}(q, t)$), then $s.p = s.q = t.q = t.p$. Otherwise, no renaming steps of the sequence transforming s to t influence position p, so $s.p = t.p$. By Theorem 3.5, $s = t$. □

6.2 Substitutivity

At this point, it is inevitable to make one more assumption about our algebra. Namely, assume in the rest that there is infinitely many variables of each type τ_b where b is any binding index. This is a standard restriction which guarantees that we can rename a bound variable with a fresh one whenever necessary. The following proposition states this in more detail.

Proposition 6.8. (i) *Let $\mathcal{Y}$ be any finite subsystem of $\mathcal{X}$ (i.e. $\mathcal{Y} = (\mathcal{Y}_\gamma : \gamma \in \Gamma)$ where $\mathcal{Y}_\gamma \subseteq \mathcal{X}_\gamma$ for each $\gamma \in \Gamma$). Then each term is α-congruent to some term in which no variables of $\mathcal{Y}$ occur binding.*

(ii) *Whenever we have a finite family of terms, there exists a family of α-congruent to them, respectively, terms such that no variable occurs both free and binding in these terms.*

(iii) *Let σ be any type-preserving mapping of variables to terms. Then each term s is α-congruent to some term t such that applying $\lfloor\sigma\rfloor$ to t is sound.*

Proof. (i) Let s be any term. As $\mathcal{Y}$ is finite, we can find a type-preserving injective on its support mapping ϱ of variables to variables with $\operatorname{supp}\varrho = \mathcal{Y}$ such that $\varrho(y)$ neither belongs to $\mathcal{Y}$ nor occurs in s for each $y \in \mathcal{Y}$. Now apply Corollary 6.2.

(ii) Take $\mathcal{Y}$ to be the system of all variables which occur free in some of the given terms. Then apply (i) for each of the terms.

(iii) Take $\mathcal{Y}$ to be the system of all variables which occur free in $\sigma(x)$ for some $x \in \operatorname{fv} s \cap \operatorname{supp}\sigma$. Then apply (i) for s. □

Lemma 6.9. *Let s be a term and pb, $b \in \mathbf{B}_a$, its binding position. Denote $x = s.pb$ and take a variable z of the same type as x not occurring in $s.pa$. Denote $t = \mathrm{ren}_{pb \to z}(s)$. Let f be any placing rule such that $\operatorname{dom} f$ does not contain binding positions and, for each $r \in \operatorname{dom} f$, x does not occur free in $f(r)$.*

(i) *If z does not occur in $f(r)$ for any $r \in \operatorname{dom} f$, then $([f](s), [f](t)) \in \mathrm{A}^r$.*

(ii) *If z does not occur free in $f(r)$ for any $r \in \operatorname{dom} f$, then $([f](s), [f](t)) \in \alpha$.*

Proof. If $\operatorname{dom} f$ contains a prefix of p, then Theorem 4.9 gives $[f](t) = [f](s)$ from which both parts follow. Assume further no prefixes of p belonging to $\operatorname{dom} f$.

(i) It suffices to prove $[f](t) = \operatorname{ren}_{pb\to z}([f](s))$ since z is fresh by conditions. Take arbitrary $q \in \operatorname{dom} f \cap \operatorname{pos} s$ and $r \in \operatorname{bu}(pb, [f](s))$. If $r < q$, then r is not a maximal position of $[f](s)$ (since also $q \in \operatorname{pos}[f](s)$) and hence cannot belong to $\operatorname{bu}(pb, [f](s))$, a contradiction. Consider the case $q \leq r$. Taking into account that q is not a prefix of p, we have that $q = r = pb$ or $pa \leq q$. The former case cannot arise because of $\operatorname{dom} f$ not containing binding positions. The case $pa \leq q$ leads to $f(q).r\big|_q = [f](s).r = x$, whereby this occurrence of x is free in $f(q)$ since otherwise r could not belong to the binding unit of pb in $[f](s)$. As this is also excluded by the conditions, we have as the only possibility that $q \asymp r$. We conclude from all this that any position of $\operatorname{dom} f \cap \operatorname{pos} s$ is divergent from any position of $\operatorname{bu}(pb, [f](s))$.

Denote by $P(r)$ the assertion "$\operatorname{dom} f \cap \operatorname{pos} s$ contains a prefix of r". Implicitly using $[f](s) = [f\big|_{\operatorname{pos} s}](s)$ and $\operatorname{pos}[z : r \in \operatorname{bu}(pb, s), \neg P(r)](s) = \operatorname{pos} s$, we have

$$\begin{aligned}
[f](t) &= [f](\operatorname{ren}_{pb\to z}(s)) \\
&= [f]\,([z : r \in \operatorname{bu}(pb, s)](s)) && \text{(by 5.7 (i))} \\
&= [f]\,([z : r \in \operatorname{bu}(pb, s), P(r)]\,([z : r \in \operatorname{bu}(pb, s), \neg P(r)](s))) && \text{(by 4.8)} \\
&= [f]\,([z : r \in \operatorname{bu}(pb, s), \neg P(r)](s)) && \text{(by 4.9)} \\
&= [f]\,([z : r \in \operatorname{bu}(pb, [f](s))](s)) && \text{(by 5.10)} \\
&= [z : r \in \operatorname{bu}(pb, [f](s))]([f](s)) && \text{(by 4.8)} \\
&= \operatorname{ren}_{pb\to z}([f](s)). && \text{(by 5.7 (i))}
\end{aligned}$$

(ii) Find a placing rule f' with $\operatorname{dom} f = \operatorname{dom} f'$ such that, for any $r \in \operatorname{dom} f$, $(f(r), f'(r)) \in \alpha$ and z does not occur in $f'(r)$. Theorem 3.5 gives $([f](s), [f'](s)) \in \alpha$ and $([f](t), [f'](t)) \in \alpha$. By (i), $([f'](s), [f'](t)) \in \mathrm{A}$, so $([f](s), [f](t)) \in \alpha$. □

Lemma 6.10. *Let s, t be terms, $(s, t) \in \mathrm{A}$. Let ϱ be any type-preserving injective on its support mapping of variables to variables such that, for any variable $x \in \operatorname{supp}\varrho$, $\varrho(x)$ occurs in neither s nor t. Then $(\lceil\varrho\rceil(s), \lceil\varrho\rceil(t)) \in \mathrm{A}$.*

Proof. Let $t = \operatorname{ren}_{pb\to z}(s)$ for binding position pb, $b \in \mathbf{B}_a$, of s, and z not occurring in $s.pa$. By Corollary 6.2, we have $(s, \lceil\varrho\rceil(s)) \in \alpha$, as well as $(t, \lceil\varrho\rceil(t)) \in \alpha$. This gives $(\lceil\varrho\rceil(s), \lceil\varrho\rceil(t)) \in \alpha$.

Prove now that $\varrho(z)$ does not occur in $\lceil\varrho\rceil(s).pa$. Suppose the contrary, i.e. $\lceil\varrho\rceil(s).par = \varrho(z)$ for some r. Denote $y = s.par$, then $z \neq y$. Dependently on whether this occurrence of y is free or not in s, we have either $\varrho(z) = s.par = y$ or $\varrho(z) = \varrho(s.par) = \varrho(y)$. Observe that neither $\varrho(y) = z$ nor $\varrho(z) = y$ is possible because of the condition about ϱ in the formulation. So the former case cannot arise. The latter leads to exactly one of y, z belonging to $\operatorname{supp}\varrho$. But this gives either $\varrho(y) = \varrho(z) = z$ or $\varrho(z) = \varrho(y) = y$, contradicting anyway.

Hence $(\lceil\varrho\rceil(t), \operatorname{ren}_{pb\to\varrho(z)}(\lceil\varrho\rceil(s))) \in \alpha$. We show actually that $\lceil\varrho\rceil(t) = \operatorname{ren}_{pb\to\varrho(z)}(\lceil\varrho\rceil(s))$. By Claim 6.7, it remains to prove that these terms have same variables at binding positions. For position pb, we have $\operatorname{ren}_{pb\to\varrho(z)}(\lceil\varrho\rceil(s)).pb = \varrho(z) = \varrho(t.pb) = \lceil\varrho\rceil(t).pb$. For arbitrary binding position $q \neq pb$, q does not belong

to the binding unit corresponding to pb. So $\mathrm{ren}_{pb \to \varrho(z)}(\lceil\varrho\rceil(s)) \cdot q = \lceil\varrho\rceil(s) \cdot q = \varrho(s \cdot q) = \varrho(t \cdot q) = \lceil\varrho\rceil(t) \cdot q$. □

Lemma 6.11. *Let $(s,t) \in \alpha$. Let $\mathcal{Y}$ be any finite subsystem of $\mathcal{X}$ such that any variable of $\mathcal{Y}$ occurs binding in neither s nor t. Then there exist terms $s = u_0, u_1, \ldots, u_n = t$ such that, for each $i = 0, \ldots, n-1$, $(u_i, u_{i+1}) \in \mathrm{A}$ and any variable of $\mathcal{Y}$ does not occur binding in u_i.*

Proof. Since $\alpha = \mathrm{A}^{rt}$, we can find terms $v_0, \ldots, v_n$ such that $v_0 = s$, $v_n = t$ and $(v_i, v_{i+1}) \in \mathrm{A}$ for all $i = 0, \ldots, n-1$. Define a type-preserving injective on its support mapping ϱ of variables to variables such that
(1) $x \in \operatorname{supp}\varrho$ iff x is in $\mathcal{Y}$ and x occurs binding in some v_i,
(2) $x \in \operatorname{supp}\varrho$ implies $\varrho(x)$ not being in $\mathcal{Y}$ and not occurring in terms $v_0, \ldots, v_n$.
Define $u_i = \lceil\varrho\rceil(v_i)$ for all $i = 0, \ldots, n-1$. Consider arbitrary variable y from $\mathcal{Y}$, suppose it occurring binding in u_i. If y occurs at the same position also in v_i then $\varrho(y) = y$ which contradicts with the choice of ϱ (item (1)). Otherwise $y = \varrho(x)$ for some $x \neq y$ which also contradicts to the choice of ϱ (item (2)). Thus variables of $\mathcal{Y}$ do not occur binding in terms u_i. Lemma 6.10 gives $(u_i, u_{i+1}) \in \mathrm{A}$ for each $i = 0, \ldots, n-1$, so we have done. □

Theorem 6.12. *Let s,t be any α-congruent terms. Let f be any placing rule such that $\operatorname{dom} f$ contains no binding positions and, for any $p \in \operatorname{dom} f \cap \operatorname{pos} s$, variables occurring free in $f(p)$ occur binding in neither s nor t. Then $([f](s), [f](t)) \in \alpha$.*

Proof. Let $\mathcal{Y}$ be the system of variables occurring free in terms $f(p)$, $p \in \operatorname{dom} f \cap \operatorname{pos} s$. By conditions, variables of $\mathcal{Y}$ occur binding in neither s nor t. Take $u_0, \ldots, u_n$ whose existence is claimed by Lemma 6.11. By Lemma 6.9, we have $([f](u_i), [f](u_{i+1})) \in \alpha$ for each $i = 0, \ldots, n-1$. Thus $([f](s), [f](t)) \in \alpha$. □

Theorem 6.13. *Let s,t be any α-congruent terms. Let σ be any type-preserving mapping of variables to terms such that applying $\lfloor\sigma\rfloor$ to both s and t is sound. Then $(\lfloor\sigma\rfloor(s), \lfloor\sigma\rfloor(t)) \in \alpha$.*

Proof. We can express $\lfloor\sigma\rfloor(s) = [f](s)$ and $\lfloor\sigma\rfloor(t) = [f](t)$ for some f since $\mathrm{fpos}_x(s) = \mathrm{fpos}_x(t)$ for all variables x. Now apply Theorem 6.12. □

Theorem 6.13 states that different instances of an α-class are equivalent w.r.t. sound application of substitution to them. (This property is called *substitutivity* of α-congruence.) This means that the frequent practice to identify α-congruent terms is mathematically justified indeed. On the other hand, it enables to define substitution as an operation on α-classes. We go this way in the next section. The following theorem is inevitable for this.

Theorem 6.14. *Let s,t be terms, $(s,t) \in \alpha$. Let σ_1, σ_2 be type-preserving mappings of variables to terms such that $(\sigma_1(x), \sigma_2(x)) \in \alpha$ for any variable x. If applying $\lfloor\sigma_1\rfloor$ to both s and t is sound, then $(\lfloor\sigma_1\rfloor(s), \lfloor\sigma_2\rfloor(t)) \in \alpha$.*

Proof. For any variable x, we have $\mathrm{fv}\,\sigma_1(x) = \mathrm{fv}\,\sigma_2(x)$ since $(\sigma_1(x), \sigma_2(x)) \in \alpha$. As no renaming steps can be made in a pure variable, each variable is α-congruent to no other term. This leads to $\mathrm{supp}\,\sigma_1 = \mathrm{supp}\,\sigma_2$. Thus applying $\lfloor\sigma_2\rfloor$ to both s and t is sound, too. Now $(\lfloor\sigma_1\rfloor(s), \lfloor\sigma_2\rfloor(s)) \in \alpha$ by Theorem 3.5 (for the proof, take $P = \mathrm{fpos}\,s$ in the formulation of Theorem 3.5). By Theorem 6.13, we have $(\lfloor\sigma_2\rfloor(s), \lfloor\sigma_2\rfloor(t)) \in \alpha$. Transitivity gives the required claim. □

7 Substitution on α-classes

We denote s/α for the α-class of term s. Note that Corollary 6.6 allows us to extend the functions pos, fpos_x and fv naturally to α-classes. It is also reasonable to speak about binding units of α-classes although the variable which occurs in the positions of a binding unit is not determined.

As each variable form a separate α-class, we make no difference between variables and their α-classes.

Definition 7.1. Let s be any α-class of terms. Let σ be any type-preserving mapping of variables to α-classes of terms. Then define $\lfloor\sigma\rfloor(s) = \lfloor\sigma\rfloor(s)/\alpha$ where $s \in s$ and σ, $\sigma(x) \in \sigma(x)$ for any variable x, are chosen in such a way that applying $\lfloor\sigma\rfloor$ to s is sound.

The value of $\lfloor\sigma\rfloor(s)/\alpha$ does not depend on the choices made in the definition due to Theorem 6.14. A strong point of this "substitution modulo α" is that substitutions apply legally to any α-class, no worrying about variable captures is needed.

Definition 7.2. Let s be a congruence class of terms and q a position.

(i) Say that q is *significant* for s iff $s \cdot q$ belongs to the same congruence class independently on the choice of $s \in s$.

(ii) If q is significant for s, then write $s \cdot q$ for the only class which contains the terms $s \cdot q$, $s \in s$.

(iii) Denote by $\mathrm{sigpos}\,s$ the set of all positions significant for s.

Lemma 7.3. *Let s be an α-class and $p \in \mathrm{pos}\,s$. Then $p \notin \mathrm{sigpos}\,s$ iff there exists a binding position $q \in \mathrm{pos}\,s$ such that p is not a proper prefix of q and p is a prefix of some $r \in \mathrm{bu}(q, s)$.*

Proof. ($\Rightarrow$) Assume $p \notin \mathrm{sigpos}\,s$. Take $s, t \in s$ such that $(s \cdot p, t \cdot p) \notin \alpha$. By Theorem 6.5, find $s = u_0, u_1, \ldots, u_n = t$ such that $(u_i, u_{i+1}) \in \mathrm{A}$ for each $i = 0, \ldots, n-1$. There must exist i such that $(u_i \cdot p, u_{i+1} \cdot p) \notin \alpha$. Take q and z such that $u_{i+1} = \mathrm{ren}_{q \to z}(u_i)$. As $u_i \cdot p \neq u_{i+1} \cdot p$, an $r \in \mathrm{bu}(q, s)$ must exist such that $p \not> \mathrm{bu}(q, s)$. It cannot be $r < p$ since r is maximal in u_i. Thus $p \leq r$. It remains to show that p is not a proper prefix of q. If it were, then $q\big|_p$ would be a binding position in both $u_i \cdot p$ and $u_{i+1} \cdot p$. We would get

$$\begin{aligned}
u_{i+1} \cdot p &= [z : r \in \mathrm{bu}(q, u_i)](u_i) \cdot p && \text{(by 5.7 (i))}\\
&= [z : r \in \mathrm{bu}(q, u_i)\big|_p](u_i \cdot p) && \text{(by 4.7 (ii))}\\
&= [z : r \in \mathrm{bu}(q\big|_p, u_i \cdot p)](u_i \cdot p) && \text{(by 5.9)}\\
&= \mathrm{ren}_{q|_p \to z}(u_i \cdot p) && \text{(by 5.7 (i))}
\end{aligned}$$

contradicting $(u_i \cdot p, u_{i+1} \cdot p) \notin \mathrm{A}$.

($\Leftarrow$) Take $s \in \boldsymbol{s}$ and denote $x = s \cdot q$. Define $t = \mathrm{ren}_{q \to z}(s)$ where z is fresh, so also $t \in \boldsymbol{s}$. Then in terms $s \cdot p$ and $t \cdot p$, variables x and z, respectively, occur at position $r|_p$. These occurrences are free because the position q binding r in s would otherwise be a proper refinement of p contradicting the assumption. Hence $(s \cdot p, t \cdot p) \notin \alpha$ giving $p \notin \mathrm{sigpos}\, \boldsymbol{s}$. □

Lemma 7.3 implies that $\mathrm{sigpos}\, \boldsymbol{s}$ can actually be determined by any $s \in \boldsymbol{s}$. As another implication, we have $\mathrm{fpos}\, \boldsymbol{s} \subseteq \mathrm{sigpos}\, \boldsymbol{s}$ for any $\boldsymbol{s}$.

Proposition 7.4. *Let $\boldsymbol{\sigma}$ be a type-preserving mapping of variables to α-classes of terms, and $\boldsymbol{s}$ be an α-class. Assume $q \in \mathrm{sigpos}\, \boldsymbol{s}$. Then $q \in \mathrm{sigpos}\, \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s})$ whereby $\lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s}) \cdot q = \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s} \cdot q)$.*

Proof. Take $s \in \boldsymbol{s}$ and σ, $\sigma(x) \in \boldsymbol{\sigma}(x)$ for each variable x, such that applying $\lfloor \sigma \rfloor$ to s is sound. Then $\lfloor \sigma \rfloor (s) \in \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s})$.

Suppose $q \notin \mathrm{sigpos}\, \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s})$. Lemma 7.3 states that there exists a binding position $q' \in \mathrm{pos}\, \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s})$ such that $q \not< q'$ and q is a prefix of some $r \in \mathrm{bu}(q', \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s}))$. If $q' \notin \mathrm{pos}\, \boldsymbol{s}$, then $p < q'$ for some $p \in \mathrm{fpos}\, \boldsymbol{s}$ by Theorem 4.5 and thus also $p < r$. Now $q \leq p$ since $q \leq r$ and $q \in \mathrm{pos}\, \boldsymbol{s}$, contradicting $q \not< q'$. Hence $q' \in \mathrm{pos}\, \boldsymbol{s}$ which gives $r \in \mathrm{bu}(q', \boldsymbol{s})$ by Proposition 5.13. But then Lemma 7.3 gives $q \notin \mathrm{sigpos}\, \boldsymbol{s}$, contradicting the assumption. Hence we have proved that $q \in \mathrm{sigpos}\, \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s})$.

By Proposition 5.14, $\lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s}) \cdot q = \lfloor \boldsymbol{\sigma}' \rfloor (\boldsymbol{s} \cdot q)$ where σ' works differently from σ on variables $x \in \mathrm{supp}\, \sigma$ only which are not free at q in s. Suppose z being such a variable. Let pb and pa be the binding occurrence and root of binding, respectively, which bind z at q in s. Suppose moreover that z occurs free at some position r in $s \cdot q$. Then pa is the longest prefix of qr being a root of binding of z in s, so $qr \in \mathrm{bu}(pb, s)$. For a fresh variable z', define $t = \mathrm{ren}_{pb \to z'}(s)$. We have clearly $qr \in \mathrm{bu}(pb, t)$ and therefore z' occurs free at r in $t \cdot q$. On one hand, $(s, t) \in \alpha$ by the choice of t. On the other hand, $(s \cdot q, t \cdot q) \notin \alpha$ since the terms have different free variables. This contradicts the significance of q for s.

So $\lfloor \boldsymbol{\sigma}' \rfloor (\boldsymbol{s} \cdot q) = \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s} \cdot q)$ as no variable on which σ and σ' work differently occurs free in $s \cdot q$. Hence $\lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s}) \cdot q \in \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s} \cdot q)/\alpha = \lfloor \boldsymbol{\sigma} \rfloor (\boldsymbol{s} \cdot q)$, we have done. □

Theorem 7.5. *Let $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2$ be any type-preserving mappings of variables to α-classes of terms. Then $\lfloor \boldsymbol{\sigma}_1 \rfloor ; \lfloor \boldsymbol{\sigma}_2 \rfloor = \lfloor \boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor \rfloor$.*

Proof. We show for any α-class $\boldsymbol{s}$ that $\lfloor \boldsymbol{\sigma}_2 \rfloor (\lfloor \boldsymbol{\sigma}_1 \rfloor (\boldsymbol{s})) = \lfloor \boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor \rfloor (\boldsymbol{s})$.

Take $p \in \mathrm{fpos}\, \boldsymbol{s}$. Then $p \in \mathrm{sigpos}\, \boldsymbol{s}$ whereby $x = \boldsymbol{s} \cdot p$ is a variable. Thus Proposition 7.4 gives $\lfloor \boldsymbol{\sigma}_1 \rfloor (\boldsymbol{s}) \cdot p = \lfloor \boldsymbol{\sigma}_1 \rfloor (\boldsymbol{s} \cdot p) = \lfloor \boldsymbol{\sigma}_1 \rfloor (x) = \boldsymbol{\sigma}_1(x)$ and $\lfloor \boldsymbol{\sigma}_2 \rfloor (\lfloor \boldsymbol{\sigma}_1 \rfloor (\boldsymbol{s})) \cdot p = \lfloor \boldsymbol{\sigma}_2 \rfloor (\lfloor \boldsymbol{\sigma}_1 \rfloor (\boldsymbol{s}) \cdot p) = \lfloor \boldsymbol{\sigma}_2 \rfloor (\boldsymbol{\sigma}_1(x)) = (\boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor)(x) = \lfloor \boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor \rfloor (x) = \lfloor \boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor \rfloor (\boldsymbol{s} \cdot p) = \lfloor \boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor \rfloor (\boldsymbol{s}) \cdot p$.

Take now $q \asymp \mathrm{fpos}\, \boldsymbol{s}$. Take $s \in \boldsymbol{s}$ and σ_1, σ_2, σ_3, $\sigma_1(x) \in \boldsymbol{\sigma}_1(x)$, $\sigma_2(x) \in \boldsymbol{\sigma}_2(x)$, $\sigma_3(x) \in (\boldsymbol{\sigma}_1 ; \lfloor \boldsymbol{\sigma}_2 \rfloor)(x)$ for every variable x, such that applying $\lfloor \sigma_1 \rfloor$ and $\lfloor \sigma_3 \rfloor$ to s, as well as $\lfloor \sigma_2 \rfloor$ to $\lfloor \sigma_1 \rfloor (s)$, is sound. So $\lfloor \sigma_2 \rfloor (\lfloor \sigma_1 \rfloor (s)) = \lfloor \sigma_2 \rfloor (\lfloor \sigma_1 \rfloor (s))/\alpha$ and $\lfloor \sigma_1 ; \lfloor \sigma_2 \rfloor \rfloor (s) = \lfloor \sigma_3 \rfloor (s)/\alpha$. Clearly $\lfloor \sigma_1 \rfloor$ and $\lfloor \sigma_3 \rfloor$ do not replace at q when

applied to s. Also $\lfloor\sigma_2\rfloor$ does not replace at q when applied to $\lfloor\sigma_1\rfloor(s)$ since even if $\lfloor\sigma_1\rfloor(s) \cdot q = s \cdot q$ were a variable, it would not be free in s and the same root of binding would bind it also in $\lfloor\sigma_1\rfloor(s)$. Hence $\lfloor\sigma_2\rfloor(\lfloor\sigma_1\rfloor(s)) \cdot q = s \cdot q = \lfloor\sigma_3\rfloor \cdot q$.

Now theorem 3.5 applies and gives the needed result. □

References

[1] Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. Journal of Functional Programming **1** (1991) 375–416

[2] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)

[3] Barendregt, H. P.: The Lambda Calculus. North-Holland (1984)

[4] Fiore, M., Plotkin, G., Turi, D.: Abstract Syntax and Variable Binding. Proceedings of the 14th Ann. IEEE Symp. in Logic in Computer Science. IEEE CS Press (1999) 193–202

[5] Gabbay, M., Pitts, A.: A New Approach to Abstract Syntax Involving Binders. Proceedings of the 14th Ann. IEEE Symp. in Logic in Computer Science. IEEE CS Press (1999) 214–224

[6] Hindley, J. R.: Basic Simple Type Theory. Cambridge University Press (1997)

[7] Nestra, H.: Handling Substitution without Induction. In: Piliére, C. (ed.): Proceedings of the ESSLLI-2000 Student Session. University of Birmingham (2000) 178–188.

[8] Nestra, H.: A Framework for Studying Substitution. In: Gyimóthy, T. (ed.): Seventh Symposium on Programming Languages and Software Tools. University of Szeged (2001) 184–198

[9] Pfenning, F., Elliott, C.: Higher-Order Abstract Syntax. The ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (1988)

[10] Rosen, B. K.: Tree-Manipulating Systems and Church-Rosser Theorems. Journal of the Association for Computing Machinery **20** (1973) 160–197

[11] Stoughton, A.: Substitution Revisited. Theoretical Computer Science **59** (1988) 317-325

[12] Sun Y.: A Framework for Binding Operators. PhD thesis, LFCS, Edinburgh (1992)

[13] Sun Y.: An algebraic generalization of Frege structures—binding algebras. Theoretical Computer Science **211** (1999) 189-232

[14] Talcott, C.: A theory of binding structures and applications to rewriting. Theoretical Computer Science **112** (1993) 99–143

Acta Cybernetica 15 (2002) 653–667.

Standardized Event Pair Based Test Generation Method Using TSS&TP

Zoltán Pap*, Zoltán Rétháti*, Róbert Horváth*, and Gusztáv Adamis*

Abstract

In the software engineering test development takes significant resources. A general method for the creation of appropriate test suites could solve the problems of the often ad-hoc and time-consuming test generation process. The recent method uses formal specifications to support systematic derivation of complete test suites. From the formal specification using a special procedure a formalized document, the so-called Test Suite Structure (TSS) and Test Pur-poses (TP) can be created. With the help of this document developers can easily, automatically implement the test suites. The TSS&TP document also enables the persons who perform the tests to under-stand the test criteria and the steps, even if they do not actually know the protocol itself. We present a thorough picture of our test derivation method and show its efficiency on the Wireless Transaction Protocol (WTP) of the Wireless Application Protocol family (WAP). During our work in the validation phase we also found some operational flaws in the protocol specification.

Keywords: Test generation methods, Formal description, Test Purpose, Test Suite Structure, Validation, WAP, WTP

1 Introduction

Conformance testing is the process for checking whether the dynamic behavior of the already implemented protocol is in conformity with the standard. There are idealized requirements of conformance testing [1]. The test should cover the whole protocol (check all the possible functional behaviors for different event sequences - this is measured by the so-called coverage value which means what percentage of the graph of the extended state space of the automaton the test verifies). The test should also check abnormal situations, observe reactions to improper events. The test suite should be so created that the elements of it, i.e. the test cases, could be executed separately. Meeting all these requirements is a big challenge for all participants of the telecommunication field. The standardization institutes, the

*Budapest University of Technology and Economics Department of Telecommunications and Telematics H-1521 Budapest, Hungary,
e-mail: {pap, rethati, horvath, adamis}@ttt-atm.ttt.bme.hu

equipment manufacturers and service providers all could benefit from test generation methods that are standardized and easy to derive.

Usually highly qualified experienced engineers carry out the test generation process. They write the tests directly in C or in formal test language, for example in Tree and Tabular Combined Notation (TTCN), without any inner step directly from the standard, which is mostly an informal textual description. This knowledge-based, time consuming work leads to high costs, has lower reliability and is not always complete. On the top of all that it begins with a long learning phase when the developers have to get deeply acquainted with the new protocol. We call this way of test creation the "traditional method" in this paper.

The other known way of test generation is the computer aided automatic method [14][5]. This has also longer historical background, but until now there have been no real public solutions or efficient algorithms to provide satisfactory coverage by reasonable amount of test cases. The generated unstructured test suite is hard to execute. These automatic methods require a full formal specification as input.

We worked out a special method for the test generation process of conformance test cases that tries to mix the advantages of both previously mentioned methods.

The formal specification provides valuable information for the protocol, and completely describes the behavior of the automata. In the course of our work we implemented the WAP WTP [9][8] in a widely used formal description language, in the Specification and Description Language (SDL). We created a test suite to it starting from the SDL description and using our new method. We rationally built up a test suite structure, in which the enhanced test purposes of systematically divided and chosen elementary test cases are put. The enhanced test purposes contain not only the textual description of the purpose of the test case, but also the main information and properties of the test case in a regular form. The result of this procedure is a regular and formal document that we call Test Suite Structure (TSS) and Test Purposes (TP). This TSS&TP document contains the necessary information of the whole test suite. As each test purpose represents the description of one test case, the TTCN, C, or other code representations can be implemented easily, automatically. We present the whole method on the WAP WTP, and compare its properties to the traditional and computer aided solutions.

In the frame of this paper in the last chapter we note that in the model of WAP WTP we found flaws, which could - under special circumstances - possibly cause problems in the operation of the protocol [11].

2 Specification and Description Language

The Specification and Description Language (SDL)[12] is a formal language, which is widely used for specifying especially telecommunications systems. The formal description technique SDL is standardized by ITUT as Recommendation Z.100. In general one can choose different approaches describing systems. SDL puts emphasis on the behavior of these entities including data flow. Other fields of describing systems are out of the scope of the language. The development of SDL started in

1972 after observing the requirements of describing different complex systems. The first version was released in 1976, and new versions followed in each four years. One of the strengths of SDL is that it is a well-accepted world standard. It is supported by the ITU-T (CCITT) and ISO, thus it can be used independently of the different companies.

The typical properties of systems that can be effectively described in SDL are:

- The types of the systems observed can be real time or interactive
- The domain of the observation can be observation of behavior or observation of architecture.
- Level of abstraction can range from overview to details.

The preciseness of the SDL description makes possible to compile the description to other lower level languages such as JAVA or C. This process can be fully automated, so it is possible to decrease the time needed for developing systems and it is also possible to guarantee the correct behavior. And the SDL diagrams fully comply with the documentation and make it possible to maintain and develop the system easily.

2.1 The SDL system

The main object in the SDL abstraction is called system. This is the formal model of an existing or planned real system. Everything not belonging to the system is called environment. System can be open or closed (it depends on whether it has connection with its environment). Such a system is a collection of SDL processes which communicate asynchronously by exchanging messages. The reception of a message may force a process to change its state. During such a state transition, the SDL process may send new messages and/or perform operations on local variables. SDL processes are combined to (sub)systems by means of block diagrams. In a block diagram, the process specifications are referenced and the communication links among the processes and between the processes and the system environment are defined. All of the processes have their own memory for storing their own variables and state information, and all of them contain a FIFO buffer of infinite length, called queue, for the incoming signals. The process reads the signals from this queue on order of coming (this does not apply to priority signals). The process takes the signal in the first position in the queue, if the process has a predefined behavior for the signal, then it reacts accordingly, otherwise the process ignores the signal and moves on to the next one.

3 The WAP WTP

3.1 The WAP

The Wireless Application Protocol is set of protocols that operate over wireless communication networks. These protocols are designed for wireless devices such

as mobile telephones, pagers, and personal digital assistants. The specifications extend mobile networking technologies (such as digital data networking standards) and Internet technologies (such as XML, URLs, scripting, and various content formats).

The WAP consists of layers that more or less correspond to the OSI layers. Each of the layers of the architecture provides services for the layers above. The WAP architecture can be seen in Figure1.

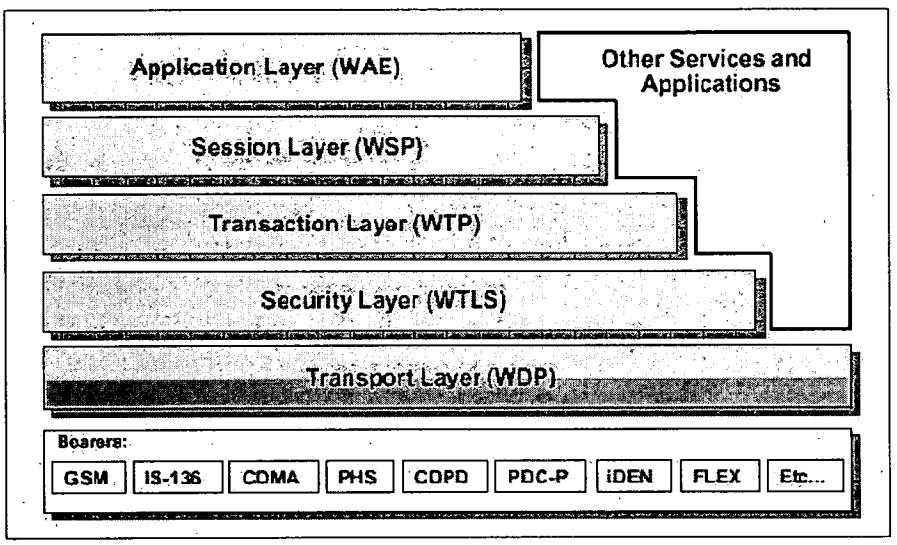


Figure 1: The WAP Architecture

3.2 The WTP

WTP is the transaction protocol of WAP. During a browsing session, the client requests information from a server, which may be fixed or mobile, and the server responds with the information. The objective of the protocol is to reliably deliver the transaction while balancing the amount of reliability required for the application with the cost of delivering the reliability. WTP runs above a datagram service and optionally a security service. The main features of the protocols are:

- Improved reliability over datagram services. WTP relieves the upper layer from re-transmissions and acknowledgements, which are necessary if datagram services are used.
- Improved efficiency over connection oriented services. WTP has no explicit connection set up.
- WTP is message oriented and designed for services oriented towards transactions, such as "browsing".

The WTP operates efficiently over secure or non-secure wireless datagram networks. There are three classes of transaction service:

- Class 0: unreliable invoke message with no result message (unreliable one-way requests),
- Class 1: reliable invoke message with no result message (reliable one-way requests),
- Class 2: reliable invoke message with exactly one reliable result message (reliable two-way request-reply).

Reliability is achieved through the use of unique transaction identifiers, acknowledgements and duplicate removal. This is more effective because explicit connection open and close makes extra load on the communication link. There is an optional user-to-user reliability: the WTP user confirms every received message. Also optional is that the last acknowledgement of the transaction may contain out of band information related to the transaction. For example, performance measurements. Concatenation may be used, where applicable, to convey multiple Protocol Data Units in one Service Data Unit of the datagram transport. The message orientation means that the basic unit of interchange is an entire message and not a stream of bytes. The transactions can be invoked at any time when needed. The protocol provides mechanisms to minimize the number of transactions being replayed as the result of duplicate packets. In case of not proper events an abort message is created and the transaction is aborted. The abort message can be sent initiated from the WTP users or initiated from the WTP providers in case of improper behavior. For reliable invoke messages, both success and failure is reported. The Responder sends back the result as the data becomes available.

4 The test generation process

During the development of a conformance test suite to a protocol several questions are arising already at the very beginning [10]. Different concepts issue in different answers. The following decisions have to be met by the test developers:

- What are the test purposes? "To analyze the right and the false behavior" theoretical purpose has to be translated into unambiguous, well-located concrete tasks. Bigger tests or "micro test cases"?
- What are the formal limits of the test cases? How to choose them, how to match them to the original test purposes? Each test case corresponds to one test purpose? Long test cases? Shorter ones?
- How to guarantee the satisfactory coverage?
- How to collect these tests into groups to support the efficient searching and overview? Storing them without any ordering in stack? Do we check the reaction to not proper events with test cases called invalid and inopportune test cases?

Of course there are the project limits: time, human and technical resources.

Even a small difference of the answers at any of these points results in a completely different test suite. As there are no central rules to have standardized tests, the test suites even for the same protocol can be completely different at different manufacturers. Later, during the interoperability tests, when the co-operation of different implementations is checked, several difficulties may come up. If there were any test or test generation procedure developed showing good performance in these fields and being recommended by a central organization, all market actors would benefit from it, and the R&D, installation and supervision costs would reduce significantly.

4.1 Computer Aided Test Generation

For some problems CATG [2] could give a solution. The time and human resource need seems to fall down to zero, and theoretically very high coverage can be reached "without work".

The experience shows a more pessimistic picture of the CATG concept [13][6]. First it can not miss the developers' active participation: the input of any test generator program has to be a formal unambiguous specification implemented in any computer language that can be processed by the program. This takes much time, and the time gain reached by CATG can not be as great as it seems to be at first sight. Moreover, creating input specification for a program needs much effort and attention, any small mistakes may lead to program errors or erroneous work. It is very hard to analyze codes generated using CATG, so the faults usually turn out only later in real life usage.

The bigger problem is the question of selecting the algorithm to be used. The NP-complete problem of state space exploration of a Communicating Extended Finite State Machine (CEFSM) makes it impossible to completely solve the computer aided test generation. The number of the generated test cases is not linearly proportional to the coverage level, as it gets close to a high percentage, and the ratio of pointless test cases increases. State space exploration algorithms derive the test cases, therefore, depending on the algorithm, two neighbor test cases may come from totally different areas of the state space. They have different length, there are no formal rules, many test cases are parts of other ones, etc.

And to top it all, the whole test suite is in an unstructured stack, and the overview or the execution of subsets according to separable protocol functions is almost impossible. There is also no solution for the proper and reasonable handling of inopportune and invalid test cases.

4.2 Atomic - event pair - test case generation through Test Suite Structure and Test Purposes

Both the traditional and the computer aided ways of test creation have strengths and weaknesses. The traditional method has the advantage of a limited set of proper and easily executable test suite, the CATG methods shorten the development time,

etc. We developed a method that takes the advantages and avoids the disadvantages of both older ones [11]. The goal was to develop a systematic way of test generation that has at least average performance in every test suite properties. In order of the points the answers in the test generation process are:

- Each test case has to verify one transition of the SDL diagram. Often, an event pair can be observed by the tester, an input and an output, so we call a single, atomic transition test case an event pair test case.
- The formal appearance and the derivation algorithm have to be common for all test cases. Each test purpose corresponds to one test case, and describes all the necessary information.
- Systematically exploring the whole SDL diagram all the "transition elements", all the event pair test cases can be created. From these elements satisfactory big test suite size can be built up, so the coverage level can meet high requirements. There are no senseless test cases, and no redundancy.
- We collect these test cases during the derivation process into test groups. The test groups identify separable functionality. The groups and the naming conventions support the easy overview and maintenance. We also create invalid and inopportune tests.

We start from a CEFSM, e.g. an SDL specification, which has to contain all the possible events and the whole automata. In contrast to the CATG method in this case the SDL diagram can have informal parts, the aim is to contain the "driver information"- the information needed to understand the functioning of the system.

To develop the whole test, the Abstract Test Suite (ATS) for a system, instead of the traditional, "directly-from-the-standard" method there are three smaller basic steps to be made. The first is to identify the test groups of the Test Suite Structure (TSS). The second is to write down the actual test cases concentrating on the test purposes - to completely define the TSS. We found a systematic procedure to derive the test cases. The last task is to implement the test cases (C code or TTCN tables), and it is almost automatic. Summary: this method consists of smaller, well localized and described tasks automated in many terms. It can be carried out with more efficiency and with less human competence, working experience and protocol behavior information. There are rules for the derivation process and formalisms that make the standardization of this method possible.

4.3 Identifying test groups

When defining the outline of the structure of the system we divide the test cases into basic groups. The question is whether a protocol parameter or variable is a test group identifier. The parameters that are set at the beginning of the test and do not change value during the test (e.g. the class identifier in a test of a classX transaction of the WTP) can be test group identifiers. During the execution of the

tests, test groups are usually run together to exhaustively test a given functionality. This is the reason to keep them in a common group.

The basic groups differ the communicating parts (e.g. Client - Server, Initiator - Responder). These groups contain the test of the parts separately, once only tests from one group have to be executed.

The different service groups are the next subgroup in the test suite. Further subgroups are formed based on other relevant features, e.g. optional services.

There are always four different standard test groups at the leaves of the test group tree of the test suite structure hierarchy.

- CA - Capability Tests. The test subgroup provides limited test of the major capabilities of the Implementation Under Test (IUT) aiming to assure that the claimed capabilities are correctly supported, in accordance with the Protocol Implementation Conformance Statement.
- BV - Valid behavior tests. The subgroup verifies that the IUT reacts in conformity with the standard, on receipt or exchange of a valid Protocol Data Unit (PDU). Valid PDU means that the exchange of messages and the content of the exchanged messages are considered as valid.
- BI - Invalid behavior tests. The subgroup verifies that the IUT is in conformity with the standard, on receipt of a syntactically invalid PDU.
- BO - Inopportune behavior tests. The test subgroup verifies that the IUT is capable of a valid reaction, when an inopportune protocol event occurs. Such an event is syntactically correct but it occurs when it is not expected.

Presentation of this step of the procedure on the WTP:

The basic groups distinguish the Initiator from the Responder side. The different transaction classes are the next subgroups (there are three services in WTP, the Class 0, 1 and 2 transaction classes). User acknowledgement is another distinct feature in the WTP. This feature is mandatory and is very important for the user applications. This is why we chose the user acknowledgement as a subgroup of our system. The test group tree of the WTP is shown in Figure 2.

4.4 Defining the Test Suite Structure

After completing the test groups the next step is to create test cases and to add them to the corresponding group. While defining the test purposes, the essential information has to be extracted from the specification. The transitions consist of four main parts:

1. Event - specifying the incoming signals.
2. Condition - that must hold to execute the Action - can also be zero Condition.
3. Action - this is to be done if both the appropriate signal arrives and the Condition holds. Tasks and output signals.
4. Next State - this is the state in which the transition leads the system.

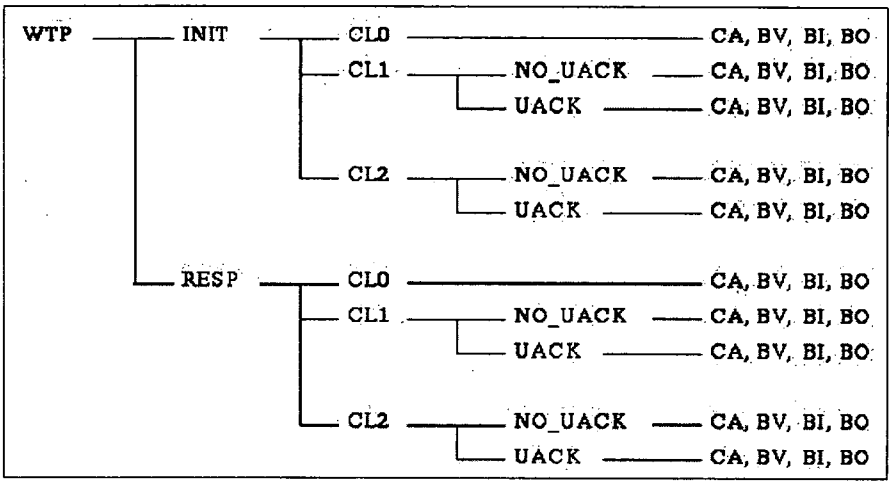


Figure 2: Test group structure of WAP WTP

When specifying test cases we set the following rules:

1. Each "normal" entry corresponds to a single test case. We call an entry a "normal" entry if it contains a single signal and no Condition.
2. Each entry with multiple Conditions corresponds to as many test cases, as many branches of the Conditions are present.
3. Each compound entry (which is for invalid and inopportune signals) corresponds to the following number of test cases:

$$Number\_of\_cases \leq \sum_{i=1}^{num\_sig} \{Sig\_errr\_fields_{sig_i}\}$$

where:

- Number_of_case: number of test cases to make,
- SIG_err_fields: number of signal fields that can contain invalid or inconsistent values,
- num_sig: number of signals that the system can normally receive during normal operation.

We systematically explore the SDL diagram. We start from the "root", the state the automaton enters after start. We describe the test cases that belong to this state, based on the branches of the SDL diagram. We create the formal Test Purpose representations (TP) and place them in the corresponding test group. After completing the test cases for one state, we go to one of the states of the next

state level in the hierarchical order of the SDL diagram, and repeat the procedure. All the test cases starting from this new state get the path of a valid test case - starting from the root and resulting in this new state - as preamble. This preamble sets the proper state of the automaton for the test case at the actual testing.

The TP in the TSS&TP document have a strictly defined appearance form. The following Table 1 pattern shows what information the TP provide about the given test cases.

TP Group	Reference
TP Id	Initial condition
	Stimulus
	Expected Behavior

Table 1: Test Purpose representation definition rules

The fields have the following meanings:

- TP Group: This shows the directory structure of the group to which the test case belongs.
- TP Id: The TP Id is a unique identifier of the test case that is specified according to naming conventions defined in the sub clause below.
- Reference: The reference should contain the references of the subject to be validated by the actual test case (specification reference, clause, paragraph).
- Initial condition: The condition defines in which initial state the IUT has to be to apply the actual test case.
- Stimulus: The stimulus defines the test event to which the test case is related.
- Expected Behavior: The expected behavior is the definition of the events that are expected from the IUT to conform to the base specification. This has to be verified by the test.

We use naming conventions in the TP definitions. The following line shows the rule of names:

Identifier: TP <fm>[<fm>...] x-<nnn>
where

TP:	Test Purpose
<fm>:	functional module
x:	type of testing (CA, BV, BO, BI)

With the help of these formula and rules we manage to make a TSS&TP document where each test purpose representation alone is also capable of telling the user immediately which part of the specification is verified by the given test case.

Presentation of this procedure step on the WTP: For the WTP we had the following test purpose naming conventions for the functional modules:

N	INITIATOR	C	CA, CAPABILITY TESTS
R	RESPONDER	V	BV, VALID BEHAV. TESTS
0	CLASS 0	O	BO,INOPORTUNE BEHAV. TESTS
1	CLASS 1	I	BI, INVALID BEHAV. TESTS
2	CLASS 2		
U	USER ACK		
E	NO USER ACK		

For example the identifier name TPR1EV-006 means: test purpose on the Responder side, Class1, no user acknowledgement, valid behavior test, and this test purpose is the 6th in the group. In Figure 3. the SDL and the test purpose representation can be seen.

The complete TSS&TP document defines the test cases and the test information, the whole dynamic behavior of the test. The test code can be easily created with the help of this document.

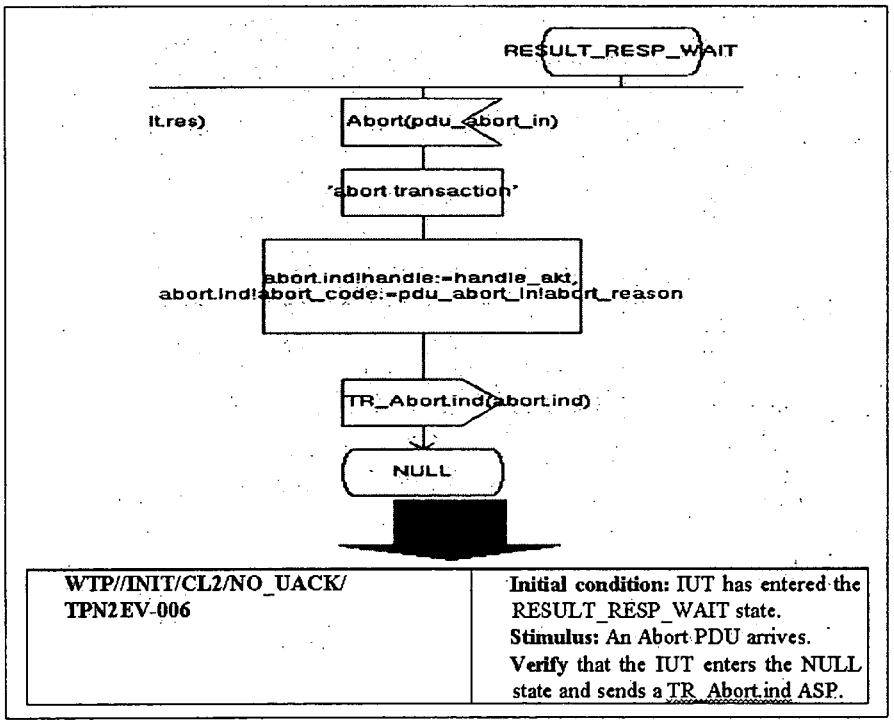


Figure 3: Derivation of a TP for a test case

4.5 Comparison to the traditional and CATG methods

Today's protocol specifications are usually based on the EFSM semantics. With the introduction of variables state space explosion may occur. So, the state space of an EFSM model can be extremely large and even infinite. Theoritically, the whole state space has to be explored and on each possible state transition a decision has to be made, if it is to be included in the test suite. This is a complex problem, that is hard to formalize. Thus, the selection of a limited appropriate set of traces – i.e. the test cases – using CATG methods, which are based on the current technology and theoritical background, fails to be useful in practice.

Our experience shows the same, the different CATG techniques resulted in an unstructured set of inefficient test cases on WAP WTP and other protocols[11]. After checking our method on these protocols and reading additional reports on CATG, we concluded the performance factors presented in Figure 4 [6][5]. The main reason for this result is easy to explain. Many parts of the test generation process can be automatized. Nevertheless, beyond a certain limit human intelligence cannot yet be substituted by pure computer based solutions. Our method tries to utilize systematic standard steps, but retains human intelligence for meeting complex decisions.

According to the previous statements, we found that our method is more practical than the traditional ones, if there is lack of highly experienced developers and the high quality of the test suite is a requirement. The Event pair TSS&TP method provides better overview, execution and easier maintenance than the CATG methods.

Method	Traditional	CATG	Event pair TSS&TP
Starting conditions	No (directly derived from the standard)	Restricted full formal specification	Formal spec. with "driver information"
Expertise needed	Very high	Medium	Medium
Time need	High	Medium	High or medium
Grouping	Incidental	No. Stack	Standardized, high
Form rules	Incidental	No	Standardized, high
Steps of the technique	One direct	Two smaller	Three smaller
Coverage	Incidental, no algorithms	High coverage by algorithms	High coverage by systematic exploration
Number of test cases	Limited	Huge at high coverage	Reasonable

Figure 4: Comparison of test generation methods

5 The checking of the model of the WAP WTP standard

In the frames of this paper we also want to note that during our work, at the verification phase [3][7] preceding the test generation process, we checked the model of the WAP WTP. Interestingly, we found flaws in the specification, which could possibly cause problems in the operation of the protocol. We validated [4] and simulated the SDL system to examine the protocol specification itself and two shortcomings arose.

5.1 Result waiting feature that may cause a problem

The first problem arises during a Class 2 transaction, if the next higher layer of the protocol on the Responder side stops its operation in a certain transaction period. In this situation both the WTP Responder and the WTP Initiator are in the state called RESULT WAIT, according to the standard. The WTP Responder is waiting for the corresponding incoming abstract service primitive (ASP) signal. If it does not come and there are no timers running, both communicating parts could wait forever for an event - that is an error on the Responder side, which inhibits the functioning of the Initiator. This is practically a deadlock, which is not resolved in this layer according to the specification. Right now the only solution could be to implement some kind of a timer in the next higher layer over the Initiator, which starts when this situation could possibly happen, and that sends an ASP signal to the Initiator after a long time without response. The first message sequent chart shows the critical situation in Figure 5.

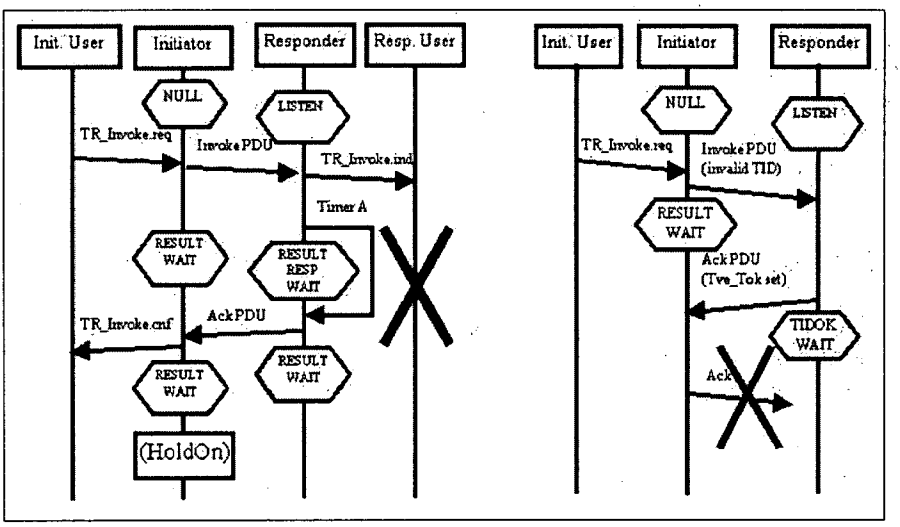


Figure 5: Critical situations in the WAP WTP model

5.2 The transaction identifier verification problem

The second problem can arise during either a Class 1 or a Class 2 transaction. The transaction identifier verification process ensures the proper sequence of the WTP PDUs. If this function has to run, and the connection between the Initiator and the Responder temporarily fails, the Responder can get stuck in the state called TIDOK WAIT. The only way to move the process out of this state is with the help of the WTP Initiator, which has to send a particular PDU signal. If this does not arrive to the Responder than it gets stuck. There is no timer that could move the process out of this state, and the next higher layer does not even get a notice about the state of the process, so a timer can not be implemented either. Even if the connection is restored, the process remains in this state. This situation can also arise, if a bad or intentionally modified Initiator implementation does not perform the right functioning. A malice communicating part can open several transactions, and leave them in deadlock in the Responder side. The second message sequent chart shows the critical situation in Figure 5.

6 Conclusion

Our test generation method consists of smaller well localized and described tasks, automated in many terms. It can be carried out with more efficiency and less human competence, working experience and protocol behavior information. There are defined formalisms and rules for the derivation process. With the help of the TSS&TP document it is now possible to test implementations of the protocol without knowing the protocol itself. It enables generating fully formal test step descriptions (for example TTCN tables) almost automatically. This test suite is more standardized, can provide high coverage, and has good or at least average performance in almost every problematic field of the test generation process.

During our work we found some flaws in the WAP WTP protocol specification, which could - under special circumstances - possibly cause problems in the operation of the protocol.

References

[1] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. *International Symposium on Software Testing and Analysis*, pages 109 – 124, 1994.

[2] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. *Automatic Test Generation for EFSM-based Systems.* http://citeseer.nj.nec.com/114451.html.

[3] EG 201 383 ETSI Guide. Methods for testing and specification (mts); use of sdl in etsi deliverables; guidelines for facilitating validation and the develop-ment of conformance tests, 1999.

[4] TCTR 004 ETSI Technical Committee Technical Report. Methods for testing and specification (mts); reports on experiments in validation methodology. Technical report, ETSI, 1996.

[5] TR 101 051 ETSI Technical Report. Methods for testing and specification (mts); report of the catg applications. Technical report, ETSI, 1999.

[6] TR 101 279 ETSI Technical Report, 1948.

[7] ETR 184 ETSI Technical Review. Methods for testing and specification (mts); over-view of validation techniques for european telecommunication standards (etss) containing sdl. Technical report, ETSI, 1995.

[8] WAP Forum. Wireless application protocol architecture specification, April 1998.

[9] WAP Forum. Wireless application protocol wireless transaction protocol specification, February 2000.

[10] B. Gregor and V. Petrenko. Protocol testing: review of methods and relevance for software testing, 1994.

[11] R. Horváth, Z. Pap, and Z. Rétháti. Methods for telecommunication protocol development and conformance testing. *Student Conference BUTE*, 2000.

[12] ITU-T. *Recommandation Z.100: Specification and Description Language*, 1992.

[13] B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt. Autolink a tool for automatic test generation from sdl specications, 1998.

[14] Gang Luo, Gregor v. Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.

Acta Cybernetica 15 (2002) 669–682.

Recognizing Design Patterns in C++ Programs with the Integration of Columbus and Maisa

Rudolf Ferenc*, Juha Gustafsson†,
László Müller‡ and Jukka Paakki§

Abstract

A method for recognizing design patterns from C++ programs is presented. The method consists of two separate phases, analysis and reverse engineering of the C++ code, and architectural pattern matching over the reverse-engineered intermediate code representation. It is shown how the pattern recognition effect can be realized by integrating two specialized software tools, the reverse engineering framework *Columbus* and the architectural metrics analyzer *Maisa*. The method and the integrated power of the tool set are illustrated with small experiments.

Keywords: design patterns, reverse engineering, source code parsing, C++, object-oriented design

1 Introduction

Due to the increase of size and complexity of software systems, the importance of being able to comprehend and assess the quality of (legacy) software code has been steadily rising. Traditional software metrics, such as complexity, cohesion, and coupling have not fully met the requirements of industrial software development, mostly because they are rather low-level concepts and do not capture the high-level design decisions actually made by the designers and programmers when constructing the software.

A more high-level view over a software system can be created by modern techniques commonly known as *reverse engineering*. In reverse engineering, the objective is to extract the static structure and the dynamic behavior of the code into some abstract representation, so as to make it easier to explore the essential aspects of the system by ignoring insignificant implementation details. In the idealistic case

*University of Szeged. Aradi Vértanuk tere 1, H-6720 Szeged, e-mail: ferenc@cc.u-szeged.hu
†University of Helsinki. P.O. Box 26, FIN-00014 Helsinki, e-mail: gustafss@cs.helsinki.fi
‡University of Szeged. Aradi Vértanuk tere 1, H-6720 Szeged, e-mail: muller_l@freemail.hu
§University of Helsinki. P.O. Box 26, FIN-00014 Helsinki, e-mail: paakki@cs.helsinki.fi

the low-level code is reverse-engineered backwards into its original design – or at least to a form that might have been the intent of the software designers.

Reverse engineering methods and tools produce a wide variety of abstract software representations. A natural and currently quite popular strategy of abstracting object-oriented programs is to extract them into a set of UML diagrams [11]. Under the assumption that UML is not just a general-purpose modeling language but also a language for describing *software architectures,* the generated diagrams can indeed be regarded as representing the architectural design of the system.

While the automatic generation of UML diagrams from software code is already supported by a number of reverse-engineering tools, it is somewhat surprising that one of the cornerstones of contemporary object-oriented software engineering, *design patterns* [5], is in almost total lack of advanced tool support. By abstracting practical solutions to frequently occurring design problems into an object-oriented format, design patterns are a most natural and useful asset when recovering the architectural design and the underlying design decisions from the software code.

In this paper we present a technique for automatically recognizing design patterns from object-oriented (C++) code. The method relies on two software tools, *Columbus* [1][3] and *Maisa* [10][12]. Columbus is a versatile reverse-engineering system that transforms C++ programs into a number of abstract representations, including UML class diagrams. Maisa is a metrics tool that analyzes the quality of a software architecture given as a set of UML diagrams. Since one of the functionalities of Maisa is the mining of design patterns from the input architecture, Columbus and Maisa together provide the combined effect of recognizing design patterns from C++ code: the code is first transformed by Columbus into UML class diagrams, which are then traversed and matched against a set of predefined design patterns by Maisa. The integration of Columbus and Maisa is technically straightforward: Columbus exports its UML diagrams into Maisa using its textual input format.

The Columbus-Maisa couple can be used both to document and analyze a software system implemented in C++. In addition to that, since the foremost application area of Maisa is the software design phase and that of Columbus is the implementation (coding) phase, the tools can be used to verify that the architectural design decisions (Maisa) are followed in the implementation phase and actually realized in the code (Columbus). This makes it possible to assess more closely the software development process as well as track the evolution of design decisions during it.

We proceed as follows. The metrics analyzer Maisa is presented in Chapter 2, concentrating especially on its pattern mining facility. The reverse-engineering system Columbus is presented in Chapter 3, followed by a short description of the tool integration in Chapter 4. In Chapter 5 we discuss our experiments on design pattern recognition. Finally, conclusions and future directions are addressed in Chapter 6.

2 Maisa

Maisa [10][12] is a software tool for the analysis of software architectures, developed in an ongoing research project at the University of Helsinki. The key idea in Maisa is to analyze design level UML diagrams and compute architectural metrics for early quality prediction of the software system.

In addition to calculating traditional (object-oriented) software metrics such as *Number of Public Methods* [2], Maisa looks for instances of design patterns (either generic ones such as the well-known GoF patterns [5] or user-defined special ones) from the UML diagrams representing the software architecture. According to the experiences gained so far with industrial cases, the level of abstraction is crucial for the success of the analysis: the more detailed the diagrams are, the more accurate are the results. Therefore design pattern mining at the detailed level of source code, as presented in this paper, is a most promising way of improving the practical usability of Maisa.

Maisa also incorporates metrics from different types of UML diagrams and execution time estimation through extended activity diagrams [15]. Additionally, we are currently studying the possibility of using dynamic information (such as sequence diagrams) for defining patterns more accurately.

2.1 Constraint satisfaction in pattern mining

Constraint satisfaction [6][7] is a generic technique that can be applied to a wide variety of tasks, in our case to mining patterns from software architectures or software code. A constraint satisfaction problem (CSP) is given as a set of variables and a set of constraints restricting the values that can be assigned to those variables. *Unary constraints* (denoted as P_i) restrict the values for a single variable, while *binary constraints* (denoted as P_{ij}) represent a condition for a pair of variables. The CSP is often modeled as a graph, where the nodes represent the variables and the arcs represent the constraints.

Formally, a CSP can be stated as follows [6]:
$(\exists x_1 \in D_1)(\exists x_2 \in D_2)...(\exists x_n \in D_n)P_1(x_1) \wedge P_2(x_2) \wedge ... \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \wedge ... \wedge P_{n-1n}(x_{n-1}, x_n)$,
with P_{ij} included for all $i < j$.

In practical terms, variable domains (D_i) must consist of a finite number of discrete values. Even so, the solution of trying out all combinations would be too slow. In addition, most combinations would make no sense, so it's no use to try them at all. We may try a particular value several times, even if there is no way that the value could be a solution for a given variable. Therefore we must find a way to effectively prune out impossible candidates.

It is not always possible or practical to find a complete solution. If we allow *partial satisfiability*, we may accept those solutions that violate (to a certain extent) some of the constraints. In this situation, the constraints do not offer just exclusive alternatives. We may define our criteria separately for each case. A disadvantage of this technique is that the number of potential solutions may go up quite rapidly.

Some research has been done regarding the case of partial satisfiability [4] and it may suit our problem quite well, as the patterns themselves are not always well-defined (discussed further in Chapter 2.3).

We define our pattern mining problem as a CSP in the following way:

- The variables (nodes) represent the roles of a pattern.
- The variable domains are initialized to contain all the names (identifiers) in the diagram(s) in question.
- Unary constraints represent conditions for a single role (e.g. the element in role X must be of type *abstract class*).
- Binary constraints represent conditions between two roles (e.g. the class in role X must be a subclass of the class in role Y).

For each pattern we compute a result, i.e. the role bindings that describe this particular pattern. The number of these bindings depends on the pattern in question. A binding is a pair {*role*,*element*}, where *role* is the name of the role and *element* is the diagram element that appears in that role, e.g. in the **Factory Method** pattern [5] two of the roles are *Product* and *Creator*.

2.2 Reducing the search space

A simple and useful way of testing the candidate values is *backtracking*, where the conditions are tested for each value. If the conditions are not met, that value is discarded. Before backtracking, we must make sure that there are no unsuitable values in the domain of each variable. This means that if we require that a certain variable can only have *class*-typed values, then we can prune all attributes, methods etc. from its domain. This way we can make the number of candidates as small as possible. Currently we use the AC-3 algorithm [6] in Maisa, but the algorithm can be easily replaced. This implementation has originally been designed by Pauli Misikangas [8].

2.2.1 AC-3 algorithm

The first and most trivial requirement is node consistency. Node i is *node consistent*, iff $\forall x \in D_i, P_i(x)$ holds. The following algorithm ensures node consistency.

```
procedure NC-1:
begin
    for i ← 1 until n do
      begin
        D_i ← {x ∈ D_i | P_i(x)}
      end
end
```

Thus, for example, all attribute-entities will be pruned by NC-1 from the domain of a variable having a constraint that allows only solutions of type *class*.

Arc consistency is defined in a similar fashion: Arc (i, j) is *arc consistent*, iff $\forall x \in D_i$ such that $P_i(x)$ holds, $\exists y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$. A more detailed discussion of arc consistency can be found in [9].

A single arc can be revised using the following procedure REVISE that returns a boolean value. The idea is similar to that behind node consistency. We delete all values from the domain of the originating node D_i, for which there are no 'legal' arcs:

```
procedure REVISE((i,j)):
begin
    DELETE ← false
        for each x ∈ D_i do
        if ∄y ∈ D_j such that P_ij(x, y), then
          begin
            delete x from D_i
            DELETE ← true
          end
    return DELETE
end
```

The AC-3 algorithm first utilizes the node consistency algorithm and then the arc consistency revision algorithm as follows. We denote the entire CSP graph with G and the respective set of arcs (constraints) with $arcs(G)$. Additionally we denote the current (non-consistent) set of arcs with Q, which means that the algorithm halts as soon as Q is empty.

```
procedure AC-3:
begin
    NC-1
    Q ← {(i, j)|(i, j) ∈ arcs(G), i ≠ j}
    while Q not empty do
      begin
        select and delete any arc (k, m) from Q
        if REVISE((k, m)) then
            Q ← Q ∪ {(i, k)|(i, k) ∈ arcs(G), i ≠ k, i ≠ m}
      end
end
```

After the domains have been made consistent, we search for correct bindings among the remaining values that satisfy the current set of constraints. In the simple case we have only one value for each variable.

2.2.2 Auxiliary facts

Many design patterns are 'related' to each other in the sense that they have common elements (see e.g. metapatterns in [13]). These relationships may be taken advantage of in two ways: the ordered search of patterns and the use of auxiliary facts [8]. When a particular pattern is being searched for, new facts are added. These facts can then be utilized later when searching for other patterns. Consider, for example, that we are searching for instances of the **Observer** pattern [5] which has both the **1:1 Connection** and the **1:N Connection** pattern [13] among its prerequisites. We would now take advantage of new facts **1:1 Connection** and **1:N Connection** that have been added while searching for instances of the respective patterns.

Most of this hierarchy consists of low-level relationships. As a consequence, we get better results by using facts extracted from source code instead of design diagrams. We can overcome some of the limitations of design diagrams (see Chapter 2.3). Nevertheless, the Maisa method is particularly well-suited to be used together with a reverse engineering tool such as Columbus.

2.3 Interaction

The AC-3 (and generally any other purely syntactic) algorithm may still produce a large number of false positives, when we have a non-trivial task like finding vaguely defined design patterns. To make matters worse, several fairly common design patterns have features that are very difficult or even impossible to model as a set of constraints. In these cases human intuition and insight is essential for verifying the potential bindings generated by the algorithm.

Many design patterns are too abstract to be easily represented syntactically [5]. The situation becomes even more complex if we require a fine-grained classification of separate pattern instances. Consider, e.g., the situation where finding instances of the metapattern **1:1 Connection** [13] is not enough, but we want to make the distinction between the patterns **Bridge** and **Command**. Their syntactic structure is alike so an attempt to automatically separate them would not be realistic.

Another related problem is that in many cases the design diagrams simply do not contain enough information. (UML) associations are a typical example. This concept has quite a lot of expressive power. An association can be implemented in a number of different ways. A common case would be to include an attribute in one of the classes containing a reference to the other class, or to have a class that calls a method of another class. During the design phase the more general representation is usually enough: either we do not know the implementation details, or we do not wish to fix them yet. However, in order to recognize some common design patterns (such as **Abstract Factory** and **Builder**), we need to know these connections explicitly. In these cases we either have to include more detailed information in the UML diagrams or try to find the patterns using incomplete information. The former alternative is not viable in practice, as in most cases we simply do not have

(or even need) the required level of detail in the design phase. As a solution to the latter case partial satisfiability techniques might be worth investigating.

Even when dealing with correct positive instances of design patterns, the number of possible bindings can become large (e.g. when searching for **Composite** or **Mediator** patterns), since the number of elements that can participate in a certain role in a pattern is not limited. The basic CSP algorithm would try to find them all. This is also a situation, where human interaction is quite helpful.

An important issue is that the rules describing the patterns are correct. This is even more important, if the semantics of the pattern are complex. Missing or false constraints may either produce a number of false positives (which can be frustrating) or false negatives (which is what we want to avoid). This issue might seem obvious, but considering the small semantical subtleties many patterns have, finding the correct representation for a pattern is not necessarily trivial.

To be of any use this kind of interaction naturally requires a highly knowledgeable user (knowledge of both the design patterns and the problem domain is essential). It must be emphasized, though, that interaction is usually not required, and the AC-3 algorithm produces results relatively fast even when working with larger domains.

Many features discussed here, such as the verification of potential bindings or the presentation of design patterns, require to extend the current user interface of Maisa. For the time being, only a textual presentation is available. In the future, more usable alternatives will be developed.

3 Columbus

Columbus is a reverse engineering framework [1][3], which has been developed in cooperation between the Research Group on Artificial Intelligence in Szeged and the Software Technology Laboratory of Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract their UML class model [11] as well as conventional call graphs.

The main motivation for developing the Columbus system has been to create a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering, and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are provided as basic parts of Columbus, while the system can be extended to serve other reverse engineering requirements as well. This way we have got a really versatile and easily extendible tool for reverse engineering.

3.1 Overview of the Columbus System

The basic operation of Columbus is performed by three types of plug-ins:

- *Extractor plug-ins* (currently an extractor for C/C++), whose task is to analyze a given input source file and to create an output file, which contains the extracted information.
- *Linker plug-ins*, whose task is to build up and filter the merged internal representation of the project. This process is carried out based on the files created by the extractor plug-in.
- *Exporter plug-ins*, whose task is to export the internal representation built up and filtered by the linker plug-in into a specific output format. (Currently: Maisa, TDE Mermaid 2.2, TED 1.0, Rational Rose, Microsoft Jet Database, HTML, XML and ASCII.)

In addition to the built-in plug-ins, the user can write and add his/her own new plug-in DLLs to the Columbus system using the *plug-in API*.

3.2 Columbus projects

The extraction process is based on the concept of a Columbus project. A project stores the input files (and their settings: precompiled header, preprocessing, output directories, message level, etc.) displayed in a tree view, which represents a real software system. The project can *simultaneously* contain source files in different programming languages. Non-source code files can be added to the project as well (e.g. documents, spreadsheets), to be displayed by Columbus using OLE technology.

3.3 The Extraction Process

The extraction process (Figure 1) itself is very similar to compilation. The first stage is data extraction. Columbus takes the input files one by one and passes them to the appropriate extractor, which creates the corresponding internal representation files. In the second stage the linker plug-in is automatically invoked in order to link (merge together) the internal representation files in the memory. In the third stage the data is transformed into a given export format, usually based on a filtered internal representation. An important advantage of Columbus is that it can incrementally perform all these steps, that is, if the partial results of certain stages are available and the input of the current stage has not been changed, the partial results will not be recreated.

3.4 CAN – The C/C++ Analyzer

Parsing of the input source code is performed by the C/C++ extractor plug-in of Columbus, which invokes a separate program called *CAN* (**C++ AN**alyzer). CAN

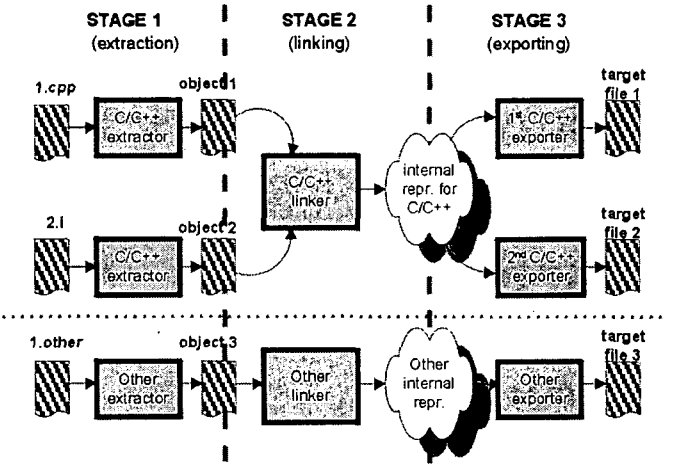


Figure 1: The extraction process

is a command-line (console) application for analyzing C/C++ code. This allows its *integration* into the user's makefiles and other configuration files, thus facilitating automated execution in parallel with the software build process.

Basically, CAN accepts one complete translation unit at a time (a preprocessed source file). For files that are not preprocessed a preprocessor will be invoked. The actual results of CAN are the internal representation files, which are the binary saves of the internal representations built up by CAN during extraction.

One of the greatest assets of CAN is probably the *handling of templates* and their *instantiation at source level*, which is accomplished using a two-pass technique in program analysis. The first pass only recognizes the language constructs in connection with the templates (like a "fuzzy" parser) and instantiates them. The second pass then performs the complete analysis of the source code and creates its internal representation.

The C++ language processed by the analyzer covers the ISO/IEC standard from 1998 [14]. Furthermore, this grammar is extended with the Microsoft extensions used in Microsoft Visual C++.

4 Integration of Columbus and Maisa

As mentioned in the previous chapter, Columbus offers an Application Programming Interface to access the information extracted from a C/C++ program. This API establishes a direct connection to the ASG (Abstract Semantic Graph) of the analyzed project, which is the common internal representation for all the information generated by the C/C++ extractor. This way it is very easy to create an

exporter plug-in for Columbus that can transform the ASG into any desired data format.

Because Maisa is implemented entirely in Java, it cannot access Columbus's ASG directly, so we have instead chosen a trivial way for connecting the two tools: an exporter plug-in in Columbus creates a file in Maisa's input file format, which can then be opened and processed further with Maisa.

The file created by Columbus contains the reverse-engineered information in PROLOG format, as facts over the main program elements (classes, attributes, etc.) and their relationships (subclassing, etc.) This information is detailed enough to support, most notably, the automatic recognition of design patterns from the underlying C++ source code.

5 Experiments

The design pattern recognition approach described above has been tested with a set of small experiments. For this purpose we have implemented some of the standard design patterns [5] in C++. After that we have used Columbus to analyze the code and to extract high-level structural information from it into the input format of Maisa. Finally, Maisa has been applied to recognize design patterns from the structural information (and, indirectly, from the original C++ code).

We demonstrate this process with the **Singleton** [5] design pattern as an example. The intent of this pattern is to ensure that a class has only one instance. One possible implementation of Singleton in C++ is as follows:

```
class MySingleton {
public:
  static MySingleton* getInstance();
protected:
  MySingleton() {};
private:
  static MySingleton* instance;
};

MySingleton* MySingleton::instance = 0;

MySingleton* MySingleton::getInstance() {
  if (instance==0) {
    instance=new MySingleton();
  }
  return instance;
}
```

The semantic intent of **Singleton** is realized by a static field that holds the only instance of the class. The constructor of this class is not accessible for other

classes. The static *getInstance* method creates the single instance, if necessary, and returns it. The only way to access the instance of the class is through this method.

When analyzing this piece of code with Columbus, we obtain (UML specific) information over class relations, such as generalizations, aggregations, associations, as well as the calling dependencies. This information is generated by Columbus into the following PROLOG-like format:

```
class("MySingleton").
method("MySingleton.getInstance()").
public("MySingleton.getInstance()").
static("MySingleton.getInstance()").
has("MySingleton","MySingleton.getInstance()").
returns("MySingleton.getInstance()","MySingleton").
method("MySingleton.MySingleton()").
protected("MySingleton.MySingleton()").
has("MySingleton","MySingleton.MySingleton()").
attribute("MySingleton.instance").
private("MySingleton.instance").
static("MySingleton.instance").
has("MySingleton","MySingleton.instance").
typeof("MySingleton.instance","MySingleton").
```

On Maisa's side, the **Singleton** pattern candidates are specified by the following facts:

```
class("Singleton").
attribute("Singleton.instance").
has("Singleton","Singleton.instance").
typeof("Singleton.instance","Singleton").
static("Singleton.instance").
```

This description states that a **Singleton** candidate (class) must have a static attribute whose type is the same as the class itself. When matching this pattern description with the high-level description of the C++ fragment, as produced by Columbus, Maisa produces the following output:

```
Solution 0
  Singleton.instance = MySingleton.instance
  Singleton = MySingleton
```

According to this, Maisa has found an instance of the **Singleton** pattern. The equations on the last two lines give the bindings generated by the AC-3 constraint satisfaction algorithm, with the name of the pattern role on the left-hand side of the equation, and the class, attribute, or method taking that role in the C++ code on the right hand side.

The following table summarizes the findings of our experiments. The table gives the names and brief descriptions of the design patterns [5] that have been recognized with the Columbus-Maisa couple.

Pattern name	Description	Missing facts
Singleton	Ensures that a class has only one instance	-
Visitor	Represents an operation on the elements of an object structure	-
Builder	Separates the creation of a complex object from its representation	reads(method,attribute) writes(method,attribute)
Factory Method	Defines an interface for creating subclass-specific objects	-
Prototype	Creates objects by cloning prototypical instances	-
Proxy	Provides a placeholder for an object to control access to it	reads(method,attribute)
Memento	Captures the state of an object	-

There are certain facts that are required for some design patterns but that Columbus does not generate yet. These facts are listed in the third column of the table. In the experiments, the additional facts were added manually to the output which was then exported to Maisa. By this, Maisa was able to correctly recognize the corresponding patterns.

The facts `reads(method,attribute)` and `writes(method,attribute)` both mean that the specified method accesses the specified attribute. The fact `writes` has the additional meaning that the state of the attribute changes in some way.

6 Conclusion and further work

We have presented a method and tool set for recognizing design patterns from C++ code. The method can be used for reverse-engineering purposes to study the structure, behavior and quality of the code, as well as for tracking the evolution of design decisions between the architectural level and the implementation level of a software system written in C++.

In our experiments it was noticed that some design patterns, like **Iterator** and **Observer**, cannot be recognized with the current method. The reason for this is that in the Maisa pattern library the descriptions of such patterns contain generated facts, i.e., structural facts that are dynamically pushed to the input by

Maisa when it recognizes a particular kind of pattern or a special kind of a common class relation. In order to recognize these kinds of design patterns, our combined method must be extended with matching of the generated facts as well.

While our initial tiny experiments show the potential capability of the pattern recognition approach, more extensive experiments with real cases must be carried out to verify the real power of the method. Such larger-scale experiments have been made with another design pattern tool [8] (using the same pattern mining algorithm as Maisa), and the results show that the technique is capable of detecting most standard design patterns quite efficiently – even those that the original programmer did not explicitly design into the code. On the other hand, it was noticed that some very abstract and fuzzy patterns (such as **Interpreter**) cannot be reliably detected by automatic means and that the performance degrades with large software systems (consisting of hundreds of thousands of program lines).

Further work is also needed for separately improving the tools. The most important improvement on the Columbus side is extending the set of generated UML diagrams beyond the currently supported class diagrams, while the main development trends in Maisa are performance analysis with extended UML activity diagrams and the use of statistical design information to predict the quality of the final system.

Acknowledgements

Maisa is being developed in a research project financed by the Finnish National Technology Agency (Tekes), Academy of Finland, Nokia Research Center, Nokia Mobile Phones, Space Systems Finland, and Kone. In addition to the Finnish co-authors of this paper, the Maisa research group includes Lilli Nenonen, Minna Majuri, and Inkeri Verkamo.

Columbus is being developed in cooperation with Nokia Research Center. In addition to the Hungarian co-authors of this paper, the Columbus research group includes Árpád Beszédes, Ferenc Magyar, and Tibor Gyimóthy.

References

[1] Beszédes, Á., Ferenc, R., Magyar, F. and Gyimóthy, T. *Columbus Setup and User's Guide.* ©1998-2000 Nokia Research Center.

[2] Chidamber, S.R., Kemerer, C.F. *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering 20,6(1994), 476-493.

[3] Ferenc, R., Magyar, F., Beszédes, Á., Kiss, Á. and Tarkiainen, M. *Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems.* In Seventh Symposium on Programming Languages and Software Tools (SPLST 2001), Szeged, Hungary, 2001, 16-27.

[4] Freuder, E., Wallace, R. *Partial Constraint Satisfaction.* Artificial Intelligence 58,1-3(1992), 21-70.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] Mackworth, A. *Consistency in Network of Relations.* Artificial Intelligence 8,1 (1977), 99-118.

[7] Mackworth, A. *The Logic of Constraint Satisfaction.* Artificial Intelligence 58,1-3 (1992), 3-20.

[8] Misikangas, P. *Automatic Recognition of Design Patterns in Object-Oriented Programs* (in Finnish). Master's Thesis C-1998-1, University of Helsinki, Department of Computer Science, 1998.

[9] Mohr, R., Henderson, T. *Arc and Path Consistency Revisited.* Artificial Intelligence, 28 (1986), 225-233.

[10] Nenonen, L., Gustafsson, J., Paakki, J. and Verkamo, A.I. *Measuring Object-Oriented Software Architectures from UML Diagrams.* In Proc. 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. Sophia Antipolis, France, 2000, 87-100.

[11] *OMG Unified Modeling Language Specification.* Version 1.3, ©1999 Object Management Group, Inc.

[12] Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L. and Verkamo, A.I. *Software Metrics by Architectural Pattern Mining.* In Proc. International Conference on Software: Theory and Practice (16th IFIP World Computer Congress). Beijing, China, 2000, 325-332.

[13] Pree, W. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[14] *Programming languages - C++.* ISO/IEC 14882:1998(E).

[15] Verkamo, A.I., Gustafsson, J., Nenonen, L. and Paakki, J. *Measuring Design Diagrams for Product Quality Evaluation.* In Proc 12th European Software Control and Metrics Conference. London, England, 2001, 357-366.

Acta Cybernetica **15** (2002) 683–704.

Development of a Communication Environment between IPv6 and IPv4

Gábor Fóris*, László Sógor†, Péter Hendlein‡, Krisztián Notaisz§ and Márta Fidrich¶

Abstract

The aim of this paper is to present the design, specification, implementation and testing of a demonstration environment for examining a genuinely new communication technique. This technique ensures that 3G mobile networks can communicate with legacy Internet phones. More than one levels of the TCP/IP protocol family are necessary for the communication, so we had to develop device drivers and user level applications too. The different levels require various development techniques and tools, whose efficiently combined usage is emphasized.

Keywords: SIP, SDP, MEGACO, IPv6, NAPT-PT

1 Motivation

The Internet Protocol version 4 (IPv4) [1] has been available since 1981. As it can be seen today, its success is indisputable. However, the rapid growth of the Internet has created a number of problems for the administration and operation of the global network, such as the quite limited address space, which will be exhausted in the near future. The new version 6 of the Internet Protocol (IPv6) [2] includes expanded addressing capabilities, header format simplification, improved support for extensions and options, plug and play services, authentication and privacy capabilities, native mobility support, and also real-time and quality services [3]. As the world continuously moves to IPv6, it is essential to solve the communication problems between the two network-layer protocols.

*GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: gabor.foris@med.ge.com

†GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: laszlo.sogor@med.ge.com

‡Axelero Tiszanet, Fekete Sas u. 28, H-6720 Szeged Hungary, e-mail: hendlein@tiszanet.hu

§GE Medical Systems, Lajos u. 48-66, H-1036 Budapest Hungary, e-mail: krisztian.notaisz@med.ge.com

¶Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanuk tere 1, H-6720 Szeged Hungary, e-mail: fidrich@sol.cc.u-szeged.hu

Since IPv6 was chosen to be the network protocol for Third Generation (3G) Mobile Networks, an urgent demand appeared in connecting the existing IPv4-based networks to the new one. Although IPv6 was designed to extend IPv4, the extension incorporates so many fundamental changes having far-reaching impacts, that IPv6 is not compatible with IPv4. It means that the IPv4-based programs need to be modified and recompiled to support IPv6 and there is no way of direct communication between IPv4-only and IPv6-only systems.

Because of the large installed base of IPv4 hosts and routers, the changeover of protocols will take still some years. During the long process of transition – which is agreed to be not easy – it is needed that programs in both realms can communicate with each other. IP-level gateways only are not enough for successful communication, since IP addresses may also be important information carried in application layer protocols, as in FTP, SIP and SDP. That is, the change of the two protocols inherently affects applications, so different level gateways (network, transport and application) have to be developed between them.

2 Networking description of the task

The 3G networks use SIP [4] (Session Initiation Protocol) as their call control protocol and IPv6 as network layer protocol for wireless communication. Currently IPv4 is used as network protocol for the Internet. There are two ways to ensure the communication (see [15]) : SIP-ALG (SIP-Application Level Gateway) and IP-level gateway remotely controlled by a SIP proxy. The remotely controlled IP-level gateway is a better solution, because rewriting the IP translator is not needed when the SIP protocol changes, it is sufficient to modify only the SIP implementations.

The goal of our work is to create a demonstration system, which connects a mobile SIP User Agent based on IPv6 and another SIP User Agent based on IPv4 via two SIP proxies (IPv6 and IPv4) and NAPT-PT [5] (Network Address Port Translator - Protocol Translator). Indeed, the real SIP communication between the IPv4 and the IPv6 networks means the communication between the two proxies via the NAPT-PT. When a media connection is created by the SIP server and client (with the help of SIP proxies), the IPv6 SIP proxy (Media Gateway Controller) sends a command using the MEGACO [7] (MEdia GAteway COntrol) protocol to add a binding to the NAPT-PT (Media Gateway). This binding helps the NAPT-PT to convert the packets transporting the media connection. When the media connection is terminated, the IPv6 SIP proxy sends a message to the NAPT-PT to release the binding. The NAPT-PT also has a DNS-ALG [8] (DNS-Application Level Gateway) extension to convert DNS queries and responses between the IPv4 and IPv6 networks.

3 Concepts of development

Due to its free source code and available documentation, the Linux operating system was chosen as a development and test platform. We worked on five computers

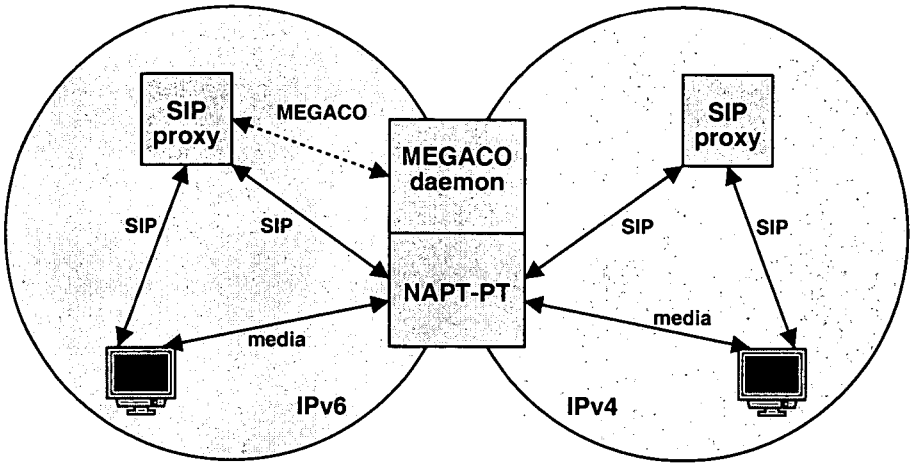


Figure 1: Test environment for multimedia call initiation between IPv4 and IPv6

having Intel Celeron processor at speed of 466 MHz using 128 MB of SDRAM at speed of 66 MHz. Each computer had two 3com 905 10/100Mb/s Ethernet cards, providing an IPv4 and an IPv4 / IPv6 interface. (Linux has dual stack IPv6 implementation, that is why we mentioned the latter interface to be dual, although we used it as a "pure" IPv6 interface.) All the PCs run GNU / Debian Linux; we relied on the last stable kernel version 2.2.17. We used the C/C++ programming languages (C for kernel level development and C++ for others).

The communication environment introduced in Section 2 requires the development of three entities, thus our work was divided into three parts: the SIP-specific NAPT-PT with DNS-ALG, the SIP proxies, and the NAPT-PT controlling MEGACO daemon were developed parallelly. Please see the table below for a summary.

part	program. level and networking layers	connections
NAPT-PT with DNS-ALG	kernel level in C network, transport and application layer	MEGACO daemon (IOCTL calls) SIP proxies (SIP commands) SIP user agents (media streams)
IPv4 SIP proxy	user level in C++ application layer	NAPT-PT (SIP commands) IPv4 SIP user agents (media & SIP)
IPv6 SIP proxy	user level in C++ application layer	NAPT-PT (SIP commands) IPv6 SIP user agents (media & SIP) MEGACO daemon (MEGACO prot.)
MEGACO daemon	user level in C++ application layer	NAPT-PT (IOCTL calls) IPv6 SIP proxy (MEGACO protocol)

1. Media Gateway
   It is a special router that knows both IPv4 and IPv6 protocols. It converts the IPv6-based TCP / UDP packages into IPv4-based packages (exchanging address and port) with the help of its inner table, and also handles SIP and DNS queries passing through. Since NAPT-PT is an IP-level packet translation mechanism and it requires kernel-level programming, we had to develop it in C programming language.

2. Media Gateway Controller
   Media connections are initiated by SIP, and it is the IPv6-based SIP proxy that controls package conversion of the NAPT-PT. SIP proxies are application-level daemons. These daemons were developed in C++ because we could use some ready-made object libraries developed for SIP over IPv4.

3. Way of Controlling
   MEGACO protocol ensures the controlling commands between NAPT-PT and IPv6-based SIP proxy at two different levels: MEGACO library aiming language recognition is used directly by the SIP proxy, while its application, the MEGACO daemon controls NAPT-PT with the help of IOCTL calls. MEGACO is a heavy protocol with highly complex grammar which we transformed into LL(k) grammar to make it consumable for the ANTLR parser generator. The object oriented programming technique is very suitable for modeling this grammar. We chose the C++ programming language because controlling of NAPT-PT requires IOCTL calls, which are available only in C/C++.

As it can be seen by now, our communication environment is a fairly complex system composed of three related yet distinct entities. Our software engineering task was not usual: we had to develop parts from kernel to user level, from network to application layer of the TCP / IP protocol family, in C and C++ languages. Indeed, communication of our software pieces required novel solutions and expert techniques. We could make good use of various development tools, whose roles are emphasized. The contribution of our work is not only the summary of a genuinely new technique to ensure communication between 3G mobile networks and legacy Internet phones. It also comes from the presentation of various software development concepts based on our consensus.

Although Internet phone programs [18] are available presently, they are based on IPv4. As far as we know, there is no software for IPv6-based phone. Also, we do not know about any implemented system (either already working or under research), which connects 3G mobile and Internet phones.

The rest of this article is organized as follows. We briefly describe commonly used tools in Section 4. After that we present the three development parts: the NAPT-PT, MEGACO and SIP. Finally, we give a conclusion, also future research is highlighted.

4 Commonly used tools

All the three groups used CVS [9] for managing the parallelly developed code. CVS is the Concurrent Versions System, the dominant open-source network-transparent version control system. CVS is useful for everyone from individual developers to large distributed teams:

- Its client-server access method lets developers access the latest code from anywhere using Internet connection.
- The management of conflicts, which can be caused by the unreserved check-out model to version control, is supported.
- Its client tools are available on most platforms.

We made good use of UML, the Unified Modeling Language at the planning of the classes. For the UML modeling we have chosen Together [10], a JAVA based development program, because it is available on various platforms, including Linux. The core features of Together are:

- Simultaneous round-trip engineering: Java, EJBs, IDL, and C++
- Multi-level, flexible documentation generation
- Multi-user team support across the enterprise
- Large project support

We needed a network traffic analyzer program for testing the conversion algorithms (NAPT-PT and DNS-ALG) and monitoring the network traffic. We chose Ethereal [16], a "sniffer" available on Unix-like operating systems. It uses GTK+, a graphical user interface library, and libpcap, a packet capture and filtering library. Ethereal knows both IPv4 and IPv6 network-layer protocols, TCP and UDP transport-layer protocols and some application-layer protocols, for example DNS and HTTP. For an Ethereal snapshot, see Fig 2.

5 NAPT-PT

5.1 Implementation form of NAPT-PT

Unlike a typical monolithic kernel, the Linux kernel has a feature to be able to load and run kernel extensions (for example device drivers) without re-linking the kernel and restarting the system. These extensions are called modules. Loading and removing of modules can be controlled manually (the system administrator may load and remove modules using `insmod`, `rmmod` and `modprobe` programs) and automatically (when the system needs it, it is loaded).

As NAPT-PT (Network Address Port Translator - Protocol Translator) is a special IP packet conversion algorithm that needs services of the kernel (functions, data structures, etc.), it has been implemented at kernel-level. We chose the module form because it is not necessary to reboot the system when we want to run our new code, it is enough to remove the old module and load the new one.

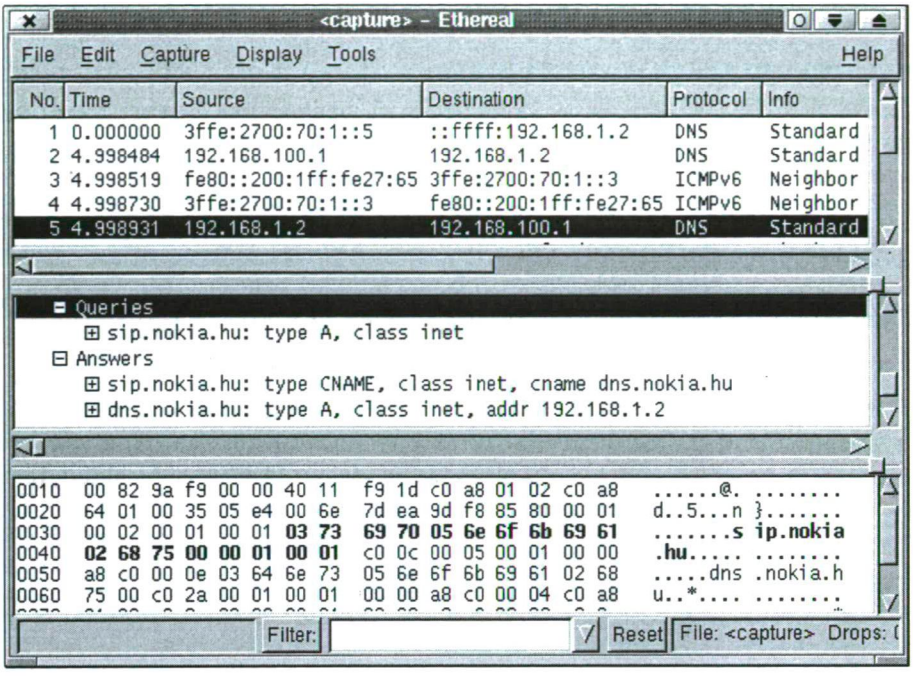


Figure 2: The Ethereal network traffic analyzer

5.2 NAPT-PT as network device

We implemented NAPT-PT as a pseudo network device, because in this way packets that needs conversion can be simply routed to it using the standard `route` procedure. Moreover, network devices can be easily configured with the `ifconfig` utility. All utilities used to manipulate pseudo network devices are part of standard Linux distributions; direct interaction between a user-level application and the loaded module is also possible with the `ioctl` system call. More details about networking in the Linux kernel can be found in [22].

5.3 Parts of NAPT-PT

Our NAPT-PT implementation consists of the following parts: packet conversion, conversion database, configuration interface and DNS-ALG.

- Packet conversion: every IPv4 and IPv6 packets that need to be converted go through this part. First this code determines the type of the packet: DNS, SIP or media stream. If the packet is a DNS one, the conversion procedure will pass it to the DNS-ALG module which translates is. The handling of a SIP packet is trivial, it does not require special algorithm just forwarding.

So as to convert a media stream packet (IPv4 packets to IPv6, and vica versa), data are needed from the conversion database. The actual IPv4 – IPv6 conversion procedure is described in our previous article [17].

- The conversion database contains records with the following fields: IPv6 address / port, IPv4 address / port and protocol id. It is important to find the record having the requested properties quickly, so we had to find a data structure which could be easily implemented at kernel level and was fast enough. The static hash of chained lists has been chosen.

- Configuration interface: configuration data (addresses of IPv6 DNS server and SIP proxy as well IPv4 address pool of NAPT-PT) must be given to NAPT-PT for the proper operation. In addition to this, IPv6 address / port / protocol triplets have to be given to NAPT-PT as a binding requests and the NAPT-PT has to response with IPv4 address / port pairs. The configuration and binding data must be given by user level processes (configuration utility and MEGACO daemon). On UNIX systems, the user-level programs can control kernel-level drivers via the IOCTL interface. The user-level process has to send the kernel a triplet mentioned before, which takes 19 bytes (16 + 2 + 1). The type of IOCTL, which is to be used for network devices, can carry 16 bytes of data, so we have to use a pointer. Another difficulty is that the kernel- and user-level memories are separated: a user-level memory cell cannot be reached form kernel-level directly and vica versa. There is only one possibility to copy data between kernel- and user-level: usage of `copy_from_user` and `copy_to_user` functions.

- DNS-ALG: this extension converts the DNS queries and responses between IPv4 and IPv6 using the configuration data of NAPT-PT. Its base functionality is to convert the DNS payload, which contains the following parts: header, questions and resource records; see [8]. The IPv4 addresses are stored in A resource records, while the IPv6 ones are stored in AAAA resource records. We assume that the length of the DNS payload is less than or equal to 512 bytes. (because of using only UDP)

  Only the IPv6 DNS server and only the IPv6 SIP proxy can be reached from the IPv4-side of the NAPT-PT using the first usable address of the IPv4 address pool of the NAPT-PT. So in the case of IPv6 $\rightarrow$ IPv4 and AAAA $\rightarrow$ A RR-conversion, the RDATA of the A RR will be first_addr (32 bit, network byte order).

  We note that source DNS question / Resource Record / domain / header means the DNS question / RR / domain / header that we convert. Destination DNS question / RR / domain / header means the result of the conversion.

  When NAPT-PT recognizes a DNS packet, it passes the DNS payload to DNS-ALG module that does the conversion. The DNS payload is a rather complex structure with lots of variable length parts.

  The payload of a DNS packet contains the following parts:

1. header, this contains some fields and numbers of queries and Resource Records (RR)
2. queries
3. answer RRs
4. authority RRs
5. additional RRs

The DNS-ALG has to convert the header, the questions and RRs.

During the planning and implementation of NAPT-PT we created the entry points of DNS-ALG, so we had to design only the conversion procedure itself. There are two main conversion functions, one for IPv6 → IPv4 and another one for IPv4 → IPv6 direction: *convert_dns_64* and *convert_dns_46.*

5.4 Restrictions

- The NAPT-PT uses an IPv4 address pool and distribute ports from this to the media connections. This pool will be set by an IPv4 network address/netmask pair.
- The network and broadcast addresses will not be distributed by NAPT-PT.
- The IPv6 DNS server and SIP proxy can be reached from IPv4 network using the first IPv4 address of the NAPT-PT's address pool, that is:
  -if its netmask is 255.255.255.255, then this IPv4 address is supposed to be used for DNS and SIP queries
  -if the netmask is 255.255.255.254(2 addresses), then we return a configurational error, because we did not get an effective address
  -otherwise, we use the first effective address (the one after the network address) for DNS/SIP queries
- The distribution goes as follows:
  -ports between 6000-65531 will be handed out for TCP and UDP connections alike
  -first the ports between 6000-65535 of the first effective address will be handed out, the the next, ... and after the last one we use the first one again.
- The handling of media packages is special:
  -in case of UDP, the first effective address/port 5999 will be the source address/port in the 6 → 4 direction; in the other direction there is no such problem.
- There is no restriction upon the IP addresses
  We do not use a table for storing the free ports. If needed, we look up a free port in the NAPT-PT table.
- In case of RTP protocol, we need a double entry and for this we will always use even ports together with the next odd one.

6 MEGACO daemon

6.1 Base problem

MEGACO protocol (MEdia GAteway COntrol) is used to control a media gateway (MG) by a media gateway controller (MGC). In our project, the NAPT-PT is a MG (converts packets of the media from IPv4 to IPv6 and vica versa) and the SIP proxy is a MGC (sends requests to NAPT-PT to reserve or free IPv4 address/port in MG). Indeed, the SIP proxy uses MEGACO protocol to control the NAPT-PT, or in other words, the communication of NAPT-PT and SIP proxy is resolved by MEGACO.

We decided to separate the language recognition and the controlling parts, because the language recognition functionality is the same in the NAPT-PT and SIP proxy as well. To do so, we first created a C/C++ library, whose basic task is the recognition of the language. It handles MEGACO messages and also provides a flexible data structure (like C++ class hierarchy). Control of the MG by a MGC is achieved with the help of this MEGACO library. The library can be used in two ways depending whether the NAPT-PT or the SIP proxy wants to use it. NAPT-PT works at the network and transport layer, so it needs a daemon at the application layer, which is the layer of the MEGACO-based communication. Thus we had to develop a MEGACO daemon as well. This daemon handles MEGACO requests & answers and controls NAPT-PT by IOCTL calls. SIP proxy is capable of application-layer communication, thus it does not need any further utility. It makes use the MEGACO library directly in its media controlling part.

6.2 Design of MEGACO library

The MEGACO library is composed of three parts:

1. The core contains a manageable data structure instead of messages having difficult grammar.

2. The message parser analyzes an arrived message and creates data structure from it.

3. The data transformer generates message from the data structure.

The most important feature of the core is to ensure the manageable data structure (C++ class hierarchy) with handler functions. Each class matches to one item of the MEGACO grammar in such a way that the hierarchy of grammar elements are modeled. These classes are needed to be filled up with given values (during message parsing) and this is the task of the handler functions. Handler functions are also used to generate messages from the data structure.

The classes are grouped in some packages like

- MegacoMain: this is the base class
- Messages: classes, which realize main MEGACO messages
- Transactions: classes, which implement MEGACO transactions
- Actions: classes implementing MEGACO actions
- Commands: classes, which implement MEGACO commands
- Descriptors: classes implementing MEGACO descriptors
- SDP: this contains the simplified SDP [6] data, because in MEGACO some grammar elements may include SDP payload.

Now let us see how to generate MEGACO message from a data structure. We obtain the message string by calling the generator function of the MegacoMessage class. The final result is a string, which will be sent in the message. Traversing the attributes of the data structure makes the generation easy: concat (in this order) the start text of an object, the attributes and the end text. If an attribute is an object, call its generator function to get the value of the object. If the attribute of an object is not a further object, we put its own specific text (for example "id=value") instead of calling the generator function of attribute.

In the message analysing part we have to decide about each message whether it is an implemented MEGACO message or not. That is, the Media Gateway (MG) – or on the other side: the Media Gateway Controller (MGC) – is able to process it (i.e. to create data structure from it) or not. Since it is the most complex (and that is why the longest) part of the library, we describe its development separately in the next subsections.

6.2.1 Parser

The base problem, we had to solve, was creating a message analyzer, which has to process all incoming messages and handle the commands these contain. Our solution relies on the core of the library, which implements an object structure by covering the graph of all recognizable messages. The analyzer itself is called by the MEGACO daemon (see later), and it has to fill the object structure that represents the construction of the received message. We decided to use a parser generator to make the system more flexible and easy-to-modify.

Every code generator needs an input grammar, which mostly published in some kind of BNF form. The Backus-Naur Form (BNF) is a convenient means for writing down the grammar of a context-free language. Augmented Backus Naur Form (ABNF) [23] differs from BNF in naming rules, repetition, alternatives, order-independence, and value-ranges. The Extended Backus-Naur Form (EBNF) [24] adds the regular expression syntax of regular languages to the BNF notation, in order to allow very compact specifications.

There are lots of parser generators (YACC, ANTLR, etc.), which differ in the expectation of the type of the input grammar and the programming language of the generated code. We chose ANTLR [14] version 1.33 – also known as PCCTS – to recognize MEGACO messages because this tool can generate C++ source code, and it expects an LL(k) grammar, given in EBNF syntax, as input.[1] The generated

[1]We note that YACC expects an LALR(1) grammar: it generates a quite fast parser; however,

source code is capable to decide whether its input is an element of the language or not — this feature is indeed used for examining the syntactical correctness of the input text.

In the input file of ANTLR we have to define tokens — for the lexical analyzer — and we have to give rules - for the syntactic analyzer. To implement a code analyzer, first a lexical analyzer is used to filter the input data and to eliminate those parts of it that will not be used further on. The valuable parts can be packed into tokens, and passed on to the syntactic analyzer, which is the parser itself. The parser contains the rules of the input grammar. These rules are identified by non-terminals, and the analyzing process consists of fitting these rules. From these non-terminals ANTLR generates functions, which can receive parameters by value or by address. There is an ability in this tool to specify actions for each recognized part of the grammar. This is useful, because we can put code into the parser directly, by invoking formerly implemented functions or adding own code written in the target programming language. (The latter one generates the proper data structure about each received message.)

6.2.2 Problems and solutions

The first task we had to solve is to convert the MEGACO grammar from ABNF (in which it is published) into EBNF required by ANTLR. ABNF has such rules that could not be easily transformed into EBNF form (eg. when a non-terminal's number of repetitions is limited). These rules were extended and further checking were implemented in the source code. In this process we also had to face how annoying to replace all the arguments signed in ABNF form with its real sequence. After finishing this part, compiling by ANTLR resulted in tremendous ambiguous rule errors. This problem occurred because the input grammar was not LL(k).

To overcome this problem, we constructed new rules and transformed the already stated rules and tokens as well to get an LL(k) grammar.

6.2.3 Resolving SDP payload

The message analyzer is composed of two parts. The MEGACO parser, which recognizes all MEGACO valid messages, and a smaller one, the SDP parser, which is invoked from the MEGACO parser when needed. SDP stands for Session Description Protocol, its roles are describing properties of multimedia sessions, for example protocol, port, origin and description. Although the grammar of the SDP protocol is much smaller and less complicated than the one of MEGACO, we had to follow the same way an "LL(k)-ization" process for SDP grammar as for MEGACO grammar.

LALR(1) means some restrictions on the input grammar (or expects transferring work from the programmer).

6.3 Design of MEGACO daemon

First, we extended the MEGACO library with network managing interface, which sends and receives MEGACO messages. According to the RFC, MEGACO has to ensure message transfer, that is: in case of successful reception, it has to acknowledge it. In case of (possible) loss, it has to resend the message and it also has to drop the repetitions. We included this functionality in the network interface, which is needed for the SIP proxy and for the MEGACO daemon (controlling NAPT-PT). This interface calls the parser in the MEGACO library and if the analysed message is wrong, that is, not an element of the language generated by the MEGACO grammar, it replies an error message. If the analysed message is accepted, the interface sends an acknowledgment.

Now, let us suppose that the incoming message is accepted, i.e. element of the language. In this case the network interface passes the message to the daemon. It interprets and processes the message: e.g. it registers a new binding in the NAPT-PT or resolves a binding. If the requested action was successful, the daemon stores / removes information on binding and makes a reply message for the network interface. Otherwise, the daemon makes an error message for the network interface.

6.4 Functionality of MEGACO daemon

We had to separate the MEGACO daemon into several independent parts. The first part is the "Listener". Listener is waiting for incoming TCP sockets, and handles them. The second part is the MEGACO library extension, the MEGACO communication unit, we call it the "'Handler". It works with an incoming socket, reads the package from it, sends an acknowledgment to it, and forwards the MEGACO data to the Processor unit. The "Processor" analyzes this data and registers the resources in NAPT-PT. The result of registration is sent back to "Handler", and data is saved to the "Central data" unit. The core part is the "Daemon". It initializes NAPT-PT and "Listener", sends registration to IPv6 SIP proxy (using a "Handler").

Figure 3 presents the above introduced units and operations between them. Dotted-line rectangles are modeling one thread in the daemon. Here is the description of communication signaled by numbered operational arrows:

1. IOCTL
2. Initialization
3. Request to MGC (eg. Service-Change command)
4. Reply from MGC (eg. reply to Service-Change command)
5. Accepting connection, creates thread (parameter: the socket)
6. Request from MGC (eg. Add command)
7. Reply to MGC (eg. reply to Add command)
8. IOCTL
9. Reads from and writes to central data

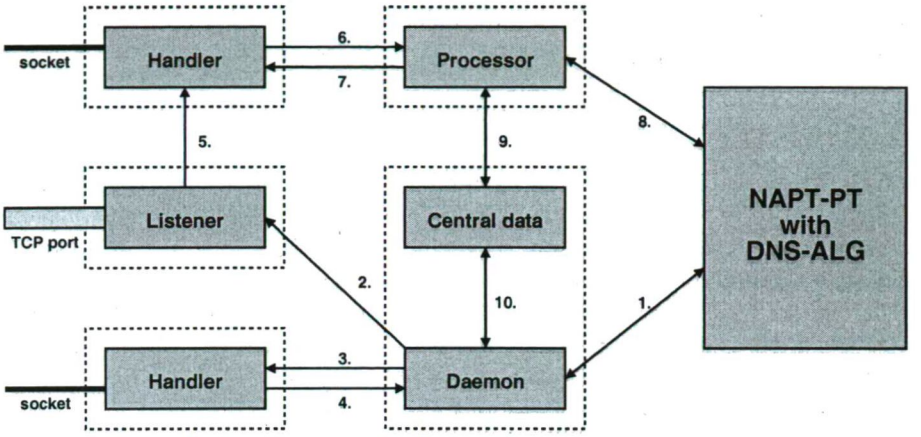


Figure 3: Main operations

10. Reads from and writes to central data

6.5 Main Restrictions

- The connection between MG (Media Gateway = NAPT-PT) and MGC (Media Gateway Controller = IPv6 SIP proxy) is based on either TCP or UDP. We have chosen TCP for the simplification of the implementation.
- We implement the text-encoding version of MEGACO.
- We formalize the grammar with PCCTS grammar analyzer, thus the not implemented commands can be inserted easily.
- We create a library, which can be used both at the implementation of MEGACO daemon and at the implementation of SIP proxy.

7 SIP proxy

The SIP protocol is used to control media sessions described by SDP [6]. We had to develop two SIP proxies, an IPv4 and an IPv6 one, the latter with MEGACO support to control the media gateway (NAPT-PT). The media connections are initiated by SIP, and the IPv6-based SIP proxy controls the NAPT-PT package conversion through adding / releasing bindings for media sessions using the MEGACO protocol.

To build the two proxies we deployed some freely available programs and libraries to reduce the development time. For example, we modified and extended

the Dissipate library – a basic SIP implementation – and Kphone [18], a SIP telephone program built on the top of Dissipate, to fit for our purposes.

7.1 Qt

Both the Dissipate library and the Kphone use the Qt library [19]. QT is a cross-platform C++ GUI framework. It provides a number of classes, for example data structures (queue, stack, list), advanced string handler functions, pattern matching functions and functions to make the building of a GUI easier.

This library indeed provided us a lot of useful classes. For example, we could make very good use of the QString class, with plenty of pattern matching functions. The parsing of the configuration file of the proxy is built on these functions. We also used some higher level data structure, like QList and QFile.

7.2 Dissipate and Kphone

Since the size of the two libraries are huge, we had to find a way to make the browsing and the understanding of the operation of the source code easy. This problem was solved by Doxygen [20]. Doxygen is a documentation system for C++, Java, IDL and C[2]. It can help you in several ways but the most useful feature of the program is that it can generate an on-line documentation (in HTML) and / or an off-line reference manual (e.g. in PDF, LaTeX) from a set of documented source files. The key to make this program really valuable is that the programmer has to put very telling comments in the source!

We also had to port the Kphone program and the Dissipate library to IPv6. This process went like this:

- Making a _v6 tagged duplicate of the original directories:
  Renaming the files and rewriting the references in the files. Browsing and rewriting the Makefiles, to enable proper compilation and installation of the Dissipate_v6 library and Kphone_v6. After this, the Kphone used the Dissipate library, the Kphone_v6 used the Dissipate_v6 library, although the code in the two libraries were yet the same.

- Locating the parts relevant to IPv6 sockets and appropriating it to IPv6 standards.

- For the cooperation with NAPT-PT and for the compatibility with RTP/RTCP [21], we had to reserve two ports for the media. However, Kphone, originally, reserves only one port.

- Correcting a bug in the Dissipate source: in extreme cases, the original code does not releases certain ports. For further details see [25].

[2]Doxygen is developed under Linux, but is set-up to be highly portable. As a result, it runs on most other UNIX flavors as well. Furthermore, an executable for Windows 9x/NT is also available.

7.3 IPv4 SIP proxy

Thinking in advance, we decided to build the IPv4 SIP proxy in a rational way to ease our work for creating the IPv6 SIP proxy. This meant that we developed a basic stateless SIP proxy to handle SIP requests and responses. Care was taken to structure the proxy in small reusable functions that might occur in other parts of the source or even in the IPv6 proxy. The proxy also makes use of the dissipate library, just as Kphone does.
We designed several classes to store and handle SIP registrations. The careful planning resulted in the following:

- The real data structure, a doubly-linked list, together with the related functions are hidden from the user of the class.
- The structure of the class makes the implementation of further functions easy.

7.4 IPv6 SIP proxy

Although, the IPv4 SIP proxy is a stateless proxy and IPv6 SIP proxy is also supposed to be stateless, the v6 version has to hold and maintain some information about SIP calls and MEGACO transactions. Furthermore, we had to use threads, because the proxy have to handle parallel MEGACO transactions and SIP calls. The connection between the two main threads (MEGACO and SIP handling) were a queue-like data structure and a state information database.

7.4.1 The functionality of SIP proxies

The v4 proxy is the base of the proxies. It was designed first and then the v6 proxy was built on top of it. Thus the functionality of the v4 proxy is a subset of the v6 proxy. A common feature is that both proxies handle registrations.

All the possible actions of the proxies are presented in Fig 4, where the meaning of the numbers are:

1. Incoming SIP message
2. Registration message
3. Registration response
4. Normal message
5. Message which does not require MEGACO
6. Message which requires MEGACO (pushed into the queue)
7. Message which requires MEGACO (popped form the queue)
8. Message with modified SDP
9. Outgoing SIP message

Note: To be precise, in case of the v4 proxy you can only have actions numbered as: 1,2,3,9 and 4,5 without the TestMegaco function.

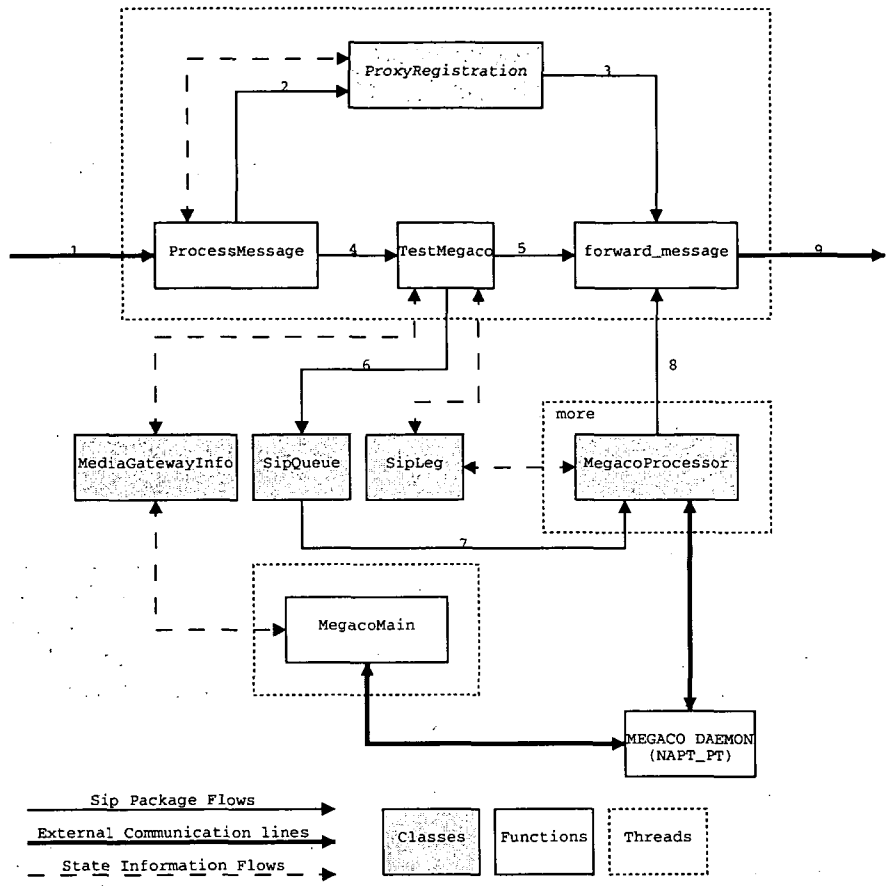


Figure 4: The functionality of SIP proxy on IPv6

7.4.2 Thread handling

Careful synchronization were needed for the reading and writing of the queue (similar to the well-known consumer / producer problem) to prevent incorrect operation and high system load. For blocking the access to data structures, mutexes and signals were used. In case of the queue, threads were put on hold while other processes read or write it, or the queue was empty. As for the state information database, it was enough to block its usage when a thread used it. We implemented some threads to clean up the data structure by erasing the expired data. One thread was necessary for waiting registrations from media gateways (MG), this was separated from the MEGACO thread that handle MEGACO transaction, because the registrations were special transactions.

7.5 Main Restrictions

- No dual-stack implementation.
- SIP server / client and the proxy must be able to handle only unicast connections, no multicast is needed.
- SIP proxy supports UDP only.
- SIP proxy is a stateless-proxy. (It stores data about the IP address conversions and the registration only.)
- Kphone supports UDP only.
- Kphone ensures RTP/RTCP compatibility.

8 Conclusion

Our study is about a novel way of connecting 3G mobile networks based on IPv6 and legacy Internet phones based on IPv4. The main idea is to connect a mobile IPv6 SIP User Agent and another SIP User Agent based on IPv4 via two SIP proxies (IPv6 and IPv4) and NAPT-PT; where the SIP communication between the IPv4 and the IPv6 networks is in fact between the two proxies via NAPT-PT. To control NAPT-PT by the IPv6 SIP proxy MEGACO protocol is used, which has good capabilities but also a highly complex syntax. NAPT-PT plays the role of Media Gateway: it converts packets of the media from IPv4 to IPv6 and vica versa; while the IPv6 SIP proxy corresponds to the Media Gateway Controller: it sends requests to NAPT-PT to reserve or free IPv6 - IPv4 address/port bindings. NAPT-PT also has a DNS-ALG extension to convert DNS queries and responses between the IPv4 and IPv6 networks.

We decided to develop a system demonstrating this communication technique. To do so, first we identified the main parts of the system: the SIP-specific NAPT-PT with DNS-ALG extension, the IPv4- and IPv6-based SIP proxies and the MEGACO daemon. After that we specified, implemented and verified these development units. Several tests were executed to find out the performance of these units, and additionally, the conformance and robustness of our implementation.

We would like to emphasize that at the time of system analysis we did not find any implementation of the previous techniques (although we could use some freely available development libraries), thus we had to develop our own pieces of software. We are still not aware of any open source implementation – which could be used for test purposes – at the time of writing. As far as we know, we are the first, who have analysed such a communication system and created a demonstration version of it.

Our communication environment is a fairly complex system composed of three related yet distinct entities. Thus our software engineering task was not usual: we had to develop parts from kernel to user level, from network to application layer of

the TCP / IP protocol family, in C and C++ languages. Indeed, communication of our software pieces required novel solutions and expert techniques. We could make good use of several development tools, whose roles are emphasized. The contribution of our work is not only the summary of a genuinely new technique, but also comes from the presentation of various software development concepts.

We have several ideas for future research in this topic. Definitely it is worth examining the case of a more general SIP User Agent capable e.g. transporting not only voice but video images thus making a real multimedia IPv6 – IPv4 session.

9 Acknowledgment

This article presents the results of a research work initiated and financed by Nokia Hungary as a joint project between University of Szeged and Nokia Hungary.

We would like to thank Gábor Bajkó, György Wolfner and László Martonossy for drawing our attention to this topic and also for their help in some technical questions. Thanks also to Dénes Bátri for his participation in the parser development and Mihály Bohus for his useful comments. Last but not least, thanks to Gergő Kiss for his valuable participation in the translator development.

10 Appendix: Configuration

10.1 Installation

Kphone is an Internet telephone program, which uses Dissipate, QT and KDE. KDE (stands for K Desktop Environment) is a free desktop system for UNIX-like systems.

Dissipate is a SIP implementation over IPv4. Kphone uses libdissipate to manage media connections. We ported libdissipate and kphone to IPv6, the name of the IPv6 library and program is libdissipate_v6 and kphone_v6.

QT is a cross-platform C++ GUI framework. The creator of QT is Trolltech (www.trolltech.com). There is an edition called 'QT/X11 Free', it is free and has available source code. The QT version 2.2.1 was used for the development, but all versions above 2.2.1 should work properly. All of the QT versions are available via FTP at 'ftp://ftp.trolltech.com/qt/source/'.

10.2 Network environment

The test environment (see Figure 5) contains 5 computers: 2 for only IPv4, 2 for only IPv6 and one for connect the IPv4 and IPv6 networks.

Network topology and IP addresses is described below:

- **v4 client:** IPv4 capable host, SIP UA
- **v4 proxy:** IPv4 capable host, IPv4 SIP proxy and master DNS server of zones '.', 'hu' and 'nokia.hu'

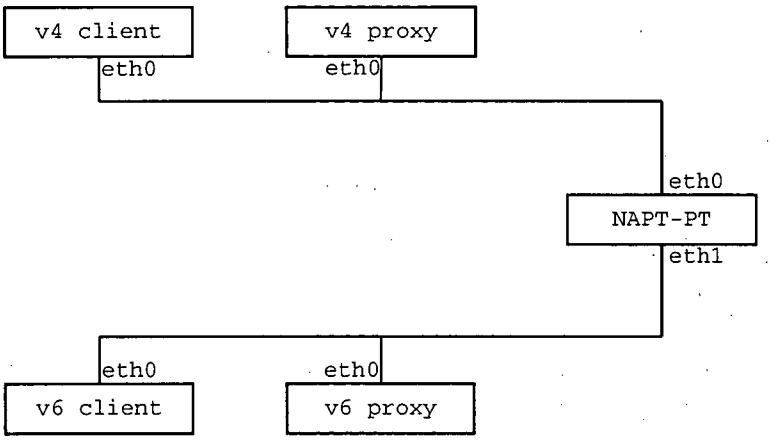


Figure 5: Network topology

- **NAPT-PT:** knows both IPv4 and IPv6 protocols, implements NAPT-PT and runs MEGACO daemon
- **v6 proxy:** IPv6 capable host, IPv6 SIP proxy, master DNS server of zone 'operator.hu' and controls NAPT-PT using MEGACO
- **v6 client:** IPv6 capable host, SIP UA

10.3 NAPT-PT

NAPT-PT is a special router that knows both IPv4 and IPv6 protocols. Because of it, NAPT-PT had to be developed in kernel level. The implementation form of the NAPT-PT is a network device. This means that you can handle it similarly to other network devices, for example the loopback (lo) or an ethernet (ethx) device.

10.3.1 Configuring the 'naptpt' network device

NAPT-PT comes as a kernel patch for Linux v2.2.17. This means you must apply this patch to the kernel. The kernel v2.2.17 can be downloaded from any official kernel mirrors, for example: ftp://ftp.[hu.]kernel.org/pub/linux/kernel/v2.2/linux-2.2.17.tar.gz

In Linux the network devices must be IPv4 and IPv6 addresses assigned to make it processing packets. We suggest that you use private, unused addresses for this purpose.

```
# ifconfig naptpt 10.0.0.1 netmask 255.255.255.255
```

Route IPv4 address pool to 'naptpt' device:

```
# route add -net 192.168.100.0 netmask 255.255.255.0 dev naptpt
```

Do the same with IPv6:

```
# ifconfig naptpt add 3ffe:ffff::1/128
```

Route the IPv4-mapped-IPv6 addresses to 'naptpt' device:

```
# route -A inet6 add ::ffff:0:0/96 dev naptpt
```

Turn on IPv4 and IPv6 packet forwardings:

```
# echo 1 >/proc/sys/net/ipv4/ip_forward
# echo 1 >/proc/sys/net/ipv6/conf/all/forwarding
```

Now, the 'naptpt' device is up and ready to work.

10.3.2 Adding/removing bindings

Adding/removing bindings is basically the task of the MEGACO daemon, but it is possible to add/remove a binding using 'naptconf' utility for testing purposes.

There are two types of bindings, single and double. To add both bindings, you need an IPv6 address/port/protocol triplet, for example: 3ffe:2700:70:1::1/13654/6, where the last component means the transport protocol(6 for TCP, 17 for UDP).

To add a single binding, you must do:

```
# naptconf 5 3ffe:2700:70:1::1 13654 6
```

If the NAPT-PT has unallocated IPv4 address/port pairs, it returns an IPv4 address/port pair, for example 192.168.100.1/6000. This means, if an IPv6 packet is sent through the NAPT-PT which has source address 3ffe:2700:70:1::1, port number 13654 and protocol number 6 (TCP), the NAPT-PT will translate it to IPv4 and the new source address/port will be 192.168.100.1/6000 with the same transport protocol than the original (TCP). The reverse direction: if an IPv4 packet is sent through the translator which has address 192.168.100.1, port 6000 and protocol number 6, it will be translated and the new IPv6 address/port pair will be 3ffe:2700:70:1::1/13654. This is really a binding between address pairs which have a given protocol number:

	addr: 3ffe:2700:70:1::1		192.168.100.1
a given protocol number:	port: 13654	———	6000
	prot: 6		6

To release this binding, you must do:

```
# naptconf 6 3ffe:2700:70:1::1 13654 6
```

Let's see the difference between a single and a double binding: a double one makes two bindings between two address/port pairs, using the previous example:

```
# naptconf 7 3ffe:2700:70:1::1 13654 6
```

The result will be two bindings:

addr: 3ffe:2700:70:1::1		192.168.100.1
port: 13654	———	6000
prot: 6		6

and

addr: 3ffe:2700:70:1::1		192.168.100.1
port: 13655	———	6001
prot: 6		6

The second binding has the almost the same properties than the first one but both IPv4 and IPv6 port numbers are increased by one.

You can add/remove double bindings using the 7/8 functions of naptconf instead of 5/6.

References

[1] J. Postel: Internet Protocol. *RFC 0791* September 1981.

[2] S. Deering and R. Hinden: Internet Protocol, Version 6 (IPv6) Specification. *RFC 2460* December 1998.

[3] C. Huitema: The new Internet Protocol. Prentice Hall 1996.

[4] M. Handley, H. Schulzrinne, E. Schooler and J. Rosenberg: SIP: Session Initiation Protocol. *RFC 2543* March 1999.

[5] G. Tsirtsis and P. Srisuresh: Network Address Translation - Protocol Translation (NAT-PT). *RFC 2766* February 2000.

[6] M. Handley and V. Jacobson: Session Description Protocol (SDP). *RFC 2327* April 1998.

[7] F. Cuervo, N. Greene, A. Rayhan, C. Huitema, B. Rosen and J. Segers: Megaco Protocol 1.0. *RFC 3015* November 2000.

[8] P. Srisuresh, G. Tsirtsis, P. Akkiraju and A. Heffernan: DNS extensions to Network Address Translators (DNS_ALG). *RFC 2694* September 1999.

[9] CVS: Concurrent Versions System, (http://www.cvshome.org/).

[10] Together is the product of TogetherSoft (http://www.togethersoft.com/).

[11] Terence John Parr: Language Translation Using PCCTS and C++ *A Reference Guide*, Automata Publishing Company, San Jose, CA 1993.

[12] Hossam Afifi, Laurent Toutain: Methods for IPv4-IPv6 transition, *The Fourth IEEE Symposium on Computers and Communications* 6 - 8 July, 1999 Red Sea, Egypt.

[13] Alain Durand: Deploying IPv6 *IEEE Internet Computing* Vol. 5, No. 1, February 2001.

[14] ANTLR: Another Tool for Language Recognition, (http://www.antlr.org/).

[15] Gábor Bajkó, Balázs Bertényi, SIP sessions between a 3G network and a SIP-proxy traversing NAT-PT (NOKIA internal report). 2000 August.

[16] Ethereal, (http://www.ethereal.com/).

[17] L. Sógor, L. Martonossy, M. Fidrich, G. Somlai, G. Dikán, P. Hendlein, T. Tarjányi and M. Bohus: Test of inter-working and translation mechanisms between IPv4 and IPv6, *Conference of PhD Students on Computer Sciences*, July 20-23, 2000, Szeged.

[18] Dissipate, a SIP implementation library and Kphone, an internet telephone program (http://www.div8.net/dissipate/).

[19] QT, the crossplatform C++ GUI framework (http://trolltech.com/).

[20] Dimitri van Heesch: Doxygen (http://www.stack.nl/~dimitri/doxygen/).

[21] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson: A Transport Protocol for Real-Time Applications (RTP). *RFC 1889* January 1996.

[22] The Linux Kernel (http://www.linux.org/).

[23] D. Crocker and P. Overell: Augmented BNF for Syntax Specifications (ABNF). *RFC 2234* November, 1997.

[24] R. S. Scowen: Extended BNF - A generic base standard (EBNF). *ISO 14977*

[25] L. Sógor, P. Hendlein, K. Notaisz, M. Fidrich, G. Fóris, G. Kiss, D. Bátri, Gy. Horváth and M. Bohus: Development of a Communication Environment between IPv6 and IPv4 *SZTE Internal Report*, April 2001, Szeged.

Acta Cybernetica **15** (2002) 705–723.

Platform Independent Tool for Local Event Correlation

Risto Vaarandi*

Abstract

Event correlation plays a crucial role in network management systems, helping to reduce the amount of event messages and making their meaning clearer to a human operator. In early network management systems, events were correlated only at network management servers. Most modern network management systems also provide means for local event correlation at agents, in order to increase the scalability of the system and to reduce network load. Unfortunately all event correlation tools currently available are commercial, quite expensive, and highly platform dependent. The author presents a free platform independent tool called *sec* for correlating network management events locally at an agent's side.

1 Introduction

Network management systems were introduced in the middle of the 1980s, in order to reduce the management costs of wide and local area networks, servers, computer applications, and services[1]. One important goal that network management systems were designed to achieve was the automation of system monitoring. A primitive example of the monitoring automation is a shell script that executes once in every 5 minutes and uses the *ping* application to check the availability of some important servers. If a server is not responding to the *ping*, the script sends an SMS-message to the system administrator's mobile phone.

Today's network management system consists of *managed objects* and *manager objects*. A managed object is a device, a computer, or an application that requires monitoring and management. A manager object is usually a dedicated *management server* that runs specific software (such as HP OpenView [7, 8] or Tivoli [2]) to perform monitoring and management tasks on managed objects. In order to monitor or manage a particular managed object, the management server contacts an *agent* that runs at the same physical node as the managed object and acts on behalf of the management server (see Figure 1).

*Department of Computer Engineering, Tallinn Technical University, Ehitajate tee 5, Tallinn 19086, Estonia, e-mail: risto.vaarandi@eyp.ee

[1]The term *network management* has become rather generic in its nature, and it is very often used to refer to a server, application, and service management as well.

Important status changes of managed objects that are observed by agents are called *events*. Examples of events are a link loss on a router, a high load on a server, or a broken TCP connection between two applications. The management server discovers new events by polling agents or by receiving notifications from them. Discovered events are presented as *event messages* to human operators at the management server's console.

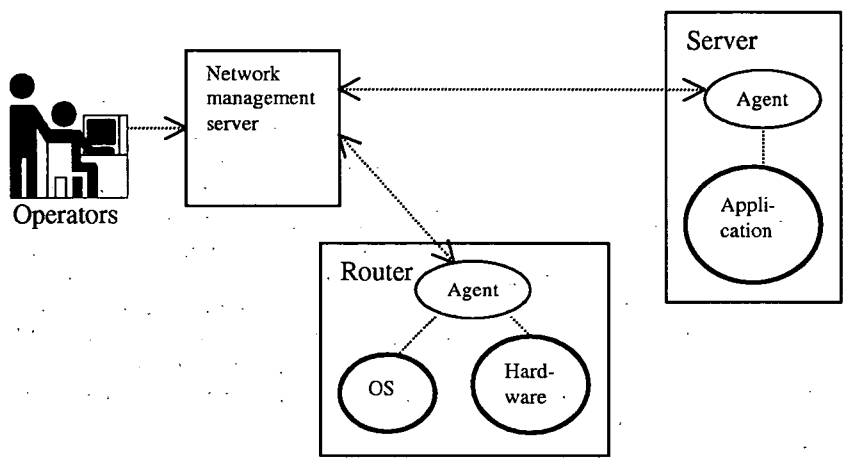


Figure 1: An example network management system (managed objects are depicted as bold circles).

If a network management system is a larger one with hundreds or thousands of managed objects, the amount of event messages often becomes too large to be handled effectively by a human. Especially undesireable situation is an *event message storm* - the flood of event messages triggered by a single hardware, software, or network failure.

In order to reduce the amount of event messages seen by the operator and to cope with various kinds of event message storms, event correlation is used. *Event correlation* is a process where irrelevant events are filtered out from being presented to a human operator and new events are derived from existing ones. For instance, events "Internal temperature of a device is too high" and "Device is unreachable" could be replaced by a single event "Device stopped working due to high internal temperature".

Most modern network management platforms provide *event correlation engines* for solving event correlation tasks [2, 6]. Since early network management systems, the network management server has been a primary location for the event correlation engine, that allows the engine to see the events of all managed objects and to correlate events even if they come from different sources.

That location of a correlation engine poses a potential threat to the scalabil-

ity of the system, since the engine processes the events of the whole system and can therefore become a bottleneck. In particular, if agents do not analyze and filter events locally and leave all that work to the correlation engine, the network management server and the engine itself could soon become overloaded. From the point of view of scalability, it would be ideal to have intelligent agents that could correlate as much events as possible locally, leaving only those events to the central correlation engine at which the knowledge of *global* context is necessary in the correlation process. Additional event correlation at an agent's side would shift load from a single network management server to many agents, reducing the possibility that the management server will become a bottleneck. Local event correlation at agents would also reduce network load, since fewer events would be reported to the network management server.

Most modern network management platforms (such as HP OpenView and Tivoli) address this problem and provide means for local event correlation [6, 7, 16].

There are two methods for correlating events locally (see Figure 2). The first method is to redirect an agent event stream (all events that go from the agent to a network management server) to a local event correlation engine, so that the network management server receives output events of the correlation engine. The other method is to correlate events *at their source*, whereby the correlation engine receives its input events from a local event source (such as application's logfile), directing its output events to the agent, while the agent event stream goes directly to the management server and remains uncorrelated. In that case the correlation engine is not located between the agent and the network management server, but between the agent and the local event source.

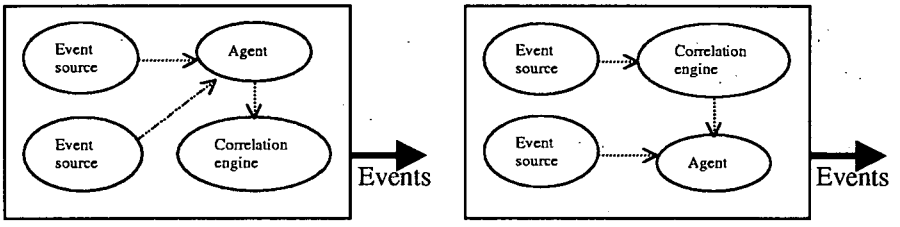


Figure 2: The methods of local event correlation.

In this paper the author presents a platform independent tool for local event correlation called *sec* (Simple Event Correlator), that implements a small set of correlation operations essential in practice. *Sec* can be used as a correlation engine for the whole agent event stream, but also for individual event sources that are files (or accessible through the file interface).

The rest of this paper is organized as follows: section 2 gives an overview of event

correlation techniques and operations; section 3 discusses related work; section 4 contains a short description of *sec*; section 5 discusses correlation rule types implemented in *sec*; section 6 describes an experiment for measuring *sec* performance; section 7 discusses the current status of *sec* and provides availability information; and section 8 concludes the paper.

2 Event correlation

Over the past ten years, many event correlation techniques have been introduced. Meira [14] provides a good overview of existing approaches.

One of the common approaches today is rule-based correlation. In the rule-based correlation, all knowledge necessary for event correlation is contained in a rule base, where each rule has the form *IF condition THEN action.* The rule-based correlation is suitable for cases where relations between events are well known and can be clearly formulated. The main disadvantage of the rule-based correlation is the lack of the learning process - past experience is not used for deriving new knowledge. This disadvantage has motivated several event correlation approaches that involve various other AI techniques, most notably neural networks. Although these techniques are quite appealing, they often suffer from the fact that their reasoning process remains unclear for the end user. For instance, neural network based computer applications sometimes produce results that are entirely unexpected even for the author of the application. However, it is essential for an AI application to reason in a manner that is clear and transparent for humans - if end users do not understand *why and how* the application reached its output, they tend to ignore the results computed by that application [15]. In the case of rule-based event correlation engines, the event correlation process is fully determined by the user-customizable rules. Since this allows the end users to fully control the work of the event correlation engine, the rule-based approach is the most accepted approach among network management engineers, and also most widely employed in today's event correlation engines [2, 4, 6, 10, 13]. In order to address the knowledge acquisition and learning problems of the rule-based approach, data mining techniques have been successfully used [11].

Jakobson and Weissman [10] provide a classification of operations that can be carried out on events during the correlation process (this classification has been adopted in a number of research papers like [11, 14]):

- Compression - substitute repeated events A with a single event A.
- Suppression - suppress an event A, if a certain operational context is present.
- Filtering - suppress an event A, if one of its parameters has a certain value.
- Counting - count repeated events A and if their number exceeds a certain threshold, replace them with a single event B.
- Scaling - in the presence of a certain operational context, replace an event A with an event B, where one of the B's parameters takes a higher value.

- Generalization - replace an event A with a more general event B, if a certain operational context is present.
- Specialization - replace an event A with a more specific event B, if a certain operational context is present.
- Temporal relationship - correlate events depending on the order of their arrival and/or the time of their generation.
- Clustering - generate an event A, if more complex correlation patterns are detected on received events. Clustering operation may take into account the result of some external tests, or combine several correlation operations listed above.

An important characteristic of an event correlation engine is its performance. Although there is no formal definition for "good performance", the event correlation engine is considered to perform well if it is able to process event floods (hundreds of input events per few seconds) in a timely manner, and if it consumes little system resources (most notably CPU time and memory). The latter condition is especially important for the local event correlation, since the local hardware resources are often quite limited (e.g., consider an event correlation task on a workstation with weak CPU and small amount of memory).

In the following section related work on event correlation is discussed.

3 Related work

Commercial network management platforms like HP OpenView and Tivoli provide network management agents that usually have some support for event correlation at event sources. The logfile monitoring module of the HP OpenView ITO agent supports compression and counting operations [7]. In the case of Tivoli Distributed Monitoring, the customer has many predefined modules that support event correlation at event sources and can develop his/her own module if desired [16].

In order to extend the capabilities of standard agents, some network management platforms provide full-featured correlation engines for agents, designed for correlating the event stream of an agent that goes to a management server. For example, HP ECS correlation engine [6] that usually runs on a network management server is also integrated into some versions of the HP OpenView ITO agent [7].

There are some obstacles to using commercial correlation engines, however. The first problem is that commercial engines are quite expensive; for instance, the price of HP ECS is around US$30,000. Many of them work only with one particular network management platform, so the customer is unable to use them independently or with other platforms.

The second problem is that commercial correlation engines have a limited support for different operating systems. For example, HP ECS is currently integrated only into HP-UX, Solaris, and Windows NT versions of the HP OpenView ITO agent. The support for different operating system platforms could be improved by

putting the source code of the correlation engine into a public domain (in order to encourage porting to other platforms), but this is not acceptable for many vendors.

Since a lot of research has been done in the field of event correlation over the past few years, some non-commercial correlation engine prototypes have been created (experimental engines presented in [5], [13], and [17] are good examples of such work). There is, however, no freeware correlation engine available yet which would be mature enough for using in a production environment. OpenNMS team that is working on the BlueBird project[2] also plans to implement the MAJI correlation engine after first versions of BlueBird have been released, but currently only the code specification of MAJI is available.

Few event correlation operations are also implemented in some freeware logfile monitoring tools. Swatch [9] supports event compression operations, allowing one to suppress repeated events in a given time frame. Logsurfer [12] is a more sophisticated tool that also supports temporal relationship operations.

One feature that almost all logfile monitors lack of is the possibility to recognize events which span over multiple logfile lines. In most logfiles each event is described by a single line, but there are still cases where one event covers two or more physical logfile lines. For example, *cron* daemons of HP-UX and Solaris write the time and the description of a single event to two consecutive lines. In addition to providing event correlation facilities for logfiles, *sec* also addresses this problem.

In the following section design goals and a short description of *sec* are presented.

4 Design goals and description of *sec*

Sec is a rule-based event correlation tool that was designed for UNIX-like operating systems. The rule-based approach was used because it is most clear and transparent for the end user (see section 2). *Sec* receives its input events from a file stream, reading new information line by line, and produces output events by executing user-specified shell commands. Design goals of *sec* were as follows:

- It should be able to handle input events regardless of their format.
- It should not be tied to any particular operating system.
- It should not be tied to any particular network management platform, but provide generic interface that makes integration with any platform possible.
- It should be possible to use it as a correlation engine for individual file-like event sources (e.g., logfiles).
- It should implement a set of correlation operations that are essential in practice, covering all operation types listed in section 2.
- It should be able to handle input events even if they arrive at a high rate.

[2] BlueBird project is an attempt to create a freeware network management platform, in order to provide a powerful and cost-effective alternative to expensive commercial platforms. See http://www.opennms.org for the current status of BlueBird and MAJI.

To be able to handle input events regardless of their format, *sec* uses regular expressions for recognizing them. Regular expressions are a natural choice because they are able to match complex patterns in input data. *Sec* also supports the use of regular expressions that match patterns spanning over multiple input lines, so input events are not restricted to have a "single line" format.

To achieve the independence from operating system platforms, the author decided to write *sec* in Perl. Perl is a widely used scripting language that runs on almost every UNIX flavour and is a standard part of many UNIX distributons. This means that applications written in Perl are able to run on a wide range of operating systems. In addition to these advantages, the support for regular expressions is integrated directly into the Perl language core. Perl programs are also almost as fast as programs written in C.

Perl 5.005 or higher is required to run *sec*, because it uses some Perl constructs for compiling regular expressions that were introduced in version 5.005. *Sec* has currently been tested on Linux, HP-UX, and Solaris, but it should run on most modern UNIX flavours.

If a regular file or standard input is specified as an input file for *sec*, *sec* acts as a correlation engine for that individual event source. In order to make the integration with arbitrary network management platform possible, sec also supports named pipes as input files. Specifying a named pipe as input is a convenient way to achieve communication between *sec* and network management agents, since the agent that supports external correlation engines can redirect its event stream from network management server to the pipe[3]. Events that the agent writes to a named pipe can have any format (and even span over multiple lines), as long as they can be matched by regular expressions.

At startup *sec* reads rules from its configuration file, opens the input and waits for new bytes to arrive. When new data become available, *sec* reads the data and updates its internal input buffer that holds N last input lines. Rules are processed then (in the same order as they were specified in the configuration file), comparing the condition part of every rule with the current content of the input buffer. After a match has been found and the action part of the rule has been executed, *sec* will optionally continue the search for new matches. The rules allow not only shell commands to be executed as actions, but they can also:

- create and delete contexts that decide whether a particular rule can be applied at the moment,
- generate new events that will serve as an input for other rules,
- reset correlation operations that are performed by other rules.

This makes it possible to combine several rules and form more complex event correlation schemes.

In the following section rule types implemented in *sec* are discussed.

[3]The author has successfully integrated *sec* with the HP OpenView ITO agent, using a named pipe for event passing.

5 Description of event correlation rule types implemented in *sec*

In the design process of *sec* the following question had to be answered - what rules should be implemented and why? Since *sec* was intended to be a tool that implements rules essential in practice, a number of event correlation guides from Cisco [1], IBM [3, 4], and HP [7, 8] were used for answering the question. Rules were derived from suggestions and example cases that were presented in more than one source document.

Rules that are currently implemented in *sec* support basic forms of *compression*, *suppression*, *filtering*, *counting*, *temporal relationship*, and *clustering* operations (see section 2). By combining several rules, many variants of every correlation operation from section 2 can be configured.

Sec configuration file consists of rule definitions, one definition per line, with whitespace-bar-whitespace string used as a field separator. Most rule definitions have the following parts:

- **Rule type** - one of the *Single*, *SingleWithScript*, *SingleWithSuppress*, *Pair*, *PairWithWindow*, *SingleWithThreshold*, *SingleWith2Thresholds*, *Suppress*, and *Calendar*.

- **Behaviour after match** - specifies whether the search for matching rules should continue after a match has been found between the current rule and input line(s). One of *TakeNext* and *DontCont* strings must be specified as a value.

- **Pattern and its type** - specifies the pattern that the input line(s) will be compared with in order to discover an event. Both regular expressions (type *RegExp*) and strings (type *SubStr*) can be used as patterns. If the type is followed by a number, the number specifies how many input lines will be used in comparison.

- **Event description** - textual description of the discovered event.

- **Action** - the action that will be executed when an event has been discovered. Actions include executing a shell command, creating a context, deleting a context, and resetting a correlation operation (e.g., reset the counting operation that is currently performed by some other rule).

- **Counting and timing constraints** - optional constraints for implementing correlation operations like counting and temporal relationship.

- **Context** - the optional context where the rule is considered valid at runtime.

In the following sections *sec* rule types are described.

5.1 Single rule

This rule does not implement any of the correlation operations but takes immediate action if certain line(s) appear in the input. It can be used as a building block for creating more complex event correlation operations. Here are some example definitions of this rule:

```
Single | DontCont | SubStr | start of maintenance |
maintenance | create

Single | DontCont | SubStr | end of maintenance |
maintenance | delete

Single | DontCont | RegExp2 | ^database error:\n(.*) |
DB error: $1 | shellcmd event.sh "%s" | !maintenance
```

The first rule creates the context *maintenance* if a line containing the substring *"start of maintenance"* appeared in the input. The second rule deletes that context if a line containing the substring *"end of maintenance"* was observed. The third rule generates an output event by executing

```
event.sh "DB error:  <error description>"
```

(%s is replaced by the event description), if a database error occurs and the application is currently not maintained (this example assumes that two consecutive lines appear in the input in the case of a database error - the line *"database error:"* followed by a line with a detailed error description).

5.2 SingleWithScript rule

This rule was designed to integrate external programs with *sec* event flow, implementing a form of the *clustering* operation. If a matching event appears in the input, an external program given by the rule definition is executed, and if the program returns zero for its exit value, then an action is executed. The definition of this rule is identical to the definition of the *Single* rule, except of the additional parameter that specifies an external program.

5.3 SingleWithSuppress rule

This rule was designed to implement one of the basic forms of the *compression* operation. If an event A is observed, the rule executes an action immediately but ignores all other instances of the event A that will appear during next t seconds. This operation is commonly needed in practice - it is implemented, for example, in HP OpenView Network Node Manager as one of the basic correlation schemes [8]. It is also provided as an example correlation scheme in Tivoli manuals [3]. In addition, rules with the same semantics are supported by logfile monitoring tools like Swatch [9] or the logfile encapsulator of HP OpenView ITO [7].

If a single hardware, software or network failure causes hundreds of events to appear in the input, this rule is useful for acting on the first matching event and ignoring all the other for a given time period. For example, if a file system becomes full, then every attempt to write to a file in that file system causes *"file system full"* message to be logged with *syslog* in many UNIX environments. If the file system that was filled up is used extensively, thousands of identical lines may be written to a logfile within few seconds, while only the first of them is important and all the other redundant[4].

Here is an example rule for handling these messages in HP-UX, that compresses all *"file system full"* events from a logfile into a single event (compression takes place for 15 minutes), and notifies local agent by calling *notify.sh* script:

```
SingleWithSuppress | DontCont | RegExp |
(\S+) [fF]ile system full | File system $1 full |
shellcmd notify.sh "%s" | 900
```

Note that *sec* considers events identical only if their descriptions are identical, so the event *"File system /home full"* will not be suppressed if it appears after *"File system /usr full"* event.

5.4 Pair rule

This rule was designed to implement one of the most common forms of the *temporal relationship* operation. If an event A is observed, the rule executes an action but ignores all other instances of the event A as long as an event B has not been observed. When the event B is observed, the rule will complete its work by executing another action. Like the previous operation, it is implemented in HP OpenView Network Node Manager as one of the basic correlation schemes [8]. It is also provided as an example in Tivoli manuals [3, 4].

This rule is useful for reducing two or more events into an event pair. A good example of application of this rule is NFS monitoring. When the NFS server becomes unreachable, the NFS client starts to log *"NFS server not responding"* messages. When the NFS server becomes reachable again, a single *"NFS server ok"* message is logged. Here is an example rule for events from a Solaris NFS client logfile that ignores redundant error events for 1 hour, producing an event both for the start and for the end of an error condition:

```
Pair | DontCont | RegExp | NFS server (\S+) not responding |
$1 is not responding | shellcmd notify.sh "%s" |
SubStr | NFS server $1 ok | $1 OK | shellcmd notify.sh "%s" | 3600
```

[4]Some implementations of syslog daemon cope with this and similar situations by implementing event compression internally.

5.5 PairWithWindow rule

This rule was designed to implement another variant of the *temporal relationship* operation. If an event A is observed, the rule waits for t seconds to see if an event B also appears (all subsequent instances of A are ignored during the waiting). If the event B is not observed within t seconds, then the action corresponding to the event A is executed; if the event B arrives on time, the action corresponding to the event B is executed.

This rule is needed when an error event becomes relevant if the error is not cleared after a certain amount of time. This is often the case for network events - in Cisco event correlation guide several common network management scenarios with Cisco devices are provided, where the presence of this rule is required for event correlation [1]. It is also provided as an example correlation scheme in Tivoli manuals [3].

Here is an example rule for events from a Cisco router logfile (the router must have *syslog* message logging enabled) that detects situations where the router does not come up within 5 minutes after an administrator has rebooted it with the *reload* command:

```
PairWithWindow | DontCont | RegExp |
(\S+) \d+: %SYS-5-RELOAD | $1 did not come up after reboot |
shellcmd notify.sh "%s" | RegExp | ($1) \d+: %SYS-5-RESTART |
$1 successful reboot | shellcmd notify.sh "%s" | 300
```

5.6 SingleWithThreshold rule

This rule was designed to implement the *counting* operation. The rule counts instances of an event A during t seconds, and if the number of events becomes equal to the threshold n before t seconds have elapsed, the rule executes an action. All subsequent instances of the event will be ignored for the rest of the time window. The time window is sliding - if less than n but more than 1 events were observed during t seconds, the beginning of the window is moved to the time moment when the second event took place.

Like the previous rule, this rule is often needed in correlating network events [1]. A rule with similar semantics is also supported by the logfile encapsulator of HP OpenView ITO [7].

This rule is useful when an event must occur repeatedly to become relevant, and a single instance of that event can be ignored. Here is an example rule for events from a Solaris *bad logins* logfile that executes *notify.sh* script, if three login failures for the same user on the same terminal were observed within 1 minute:

```
SingleWithThreshold | DontCont | RegExp | (\S+):(\S+): |
Repeated login failures for user $1 at tty $2 |
shellcmd notify.sh "%s" | 60 | 3
```

5.7 SingleWith2Thresholds rule

This rule was designed to implement another variant of the *counting* operation. The rule counts instances of an event A during t seconds and executes an action if a given threshold n is exceeded (exactly like the previous rule does). What is different from the previous rule is that the counting continues after the action has been executed - if no more than n' events A will be observed during t' seconds, the rule will execute another action. Both time windows are sliding.

This rule is useful when both the start and the end of the error condition can be discovered by counting. Here is an example rule for events from a Cisco router logfile that detects CPU overload conditions (two SYS-3-CPUHOG messages are logged within 1 minute) and also generates an event when CPU load is normal again (no SYS-3-CPUHOG messages are observed during 15 minutes):

```
SingleWith2Thresholds | DontCont | RegExp |
(\S+) \d+: %SYS-3-CPUHOG | $1 CPU overload |
shellcmd notify.sh "%s" | 60 | 2 | $1 CPU load normal |
shellcmd notify.sh "%s" | 900 | 0
```

5.8 Suppress rule

This rule was designed to implement *suppression* and *filtering* operations. Here is an example of the suppression operation (if the context *mycontext* is present, suppress all *"file system full"* events):

```
Suppress | RegExp | (\S+) [fF]ile system full | mycontext
```

Here is an example of the filtering operation (all file systems belonging to the volume group *vg01* are filtered out from being reported as full):

```
Suppress | RegExp | /dev/vg01/(\S+) [fF]ile system full
```

5.9 Calendar rule

This rule was designed for executing actions at specific times. Unlike all other rules, this rule reacts only to the system clock, ignoring other input. Time moments when the rule must act are specified in *crontab*-style. This rule can be used as a building block for creating more complex time-related event correlation operations, or for other time-related purposes.

The following example rule creates the context *NightContext* every day at 11PM; the context has a lifetime of 9 hours:

```
Calendar | 0 23 * * * | NightContext | create 32400 %s
```

5.10 Other variants of correlation operations

The rules that were described in the previous sections implemented some basic forms of *compression*, *suppression*, *filtering*, *counting*, *temporal relationship*, and *clustering* correlation operations. Other variants of correlation operations can be configured by combining several rules with appropriate actions.

Here is an example of the *specialization* operation that generates an error event if the application failed to process three subsequent incoming queries. If this was caused by a high CPU load, a more specific error event is generated.

```
# Create the context "cpu_overload" if the CPU load is
# too high, and delete it if the load is normal again

Pair | DontCont | SubStr | CPU load is too high |
cpu_overload | create | SubStr | CPU load is normal |
cpu_overload | delete | 86400

# Count the number of failed queries, and generate an
# event "3 subsequent failed queries" if the threshold
# is exceeded

SingleWithThreshold | DontCont | SubStr |
query processing failed | 3 subsequent failed queries |
event | 600 | 3

# Reset the counting done by the previous rule if some
# query is processed successfully (event text "3
# subsequent failed queries" is used to refer to the
# pending counting operation)

Single | DontCont | SubStr | query processing successful |
3 subsequent failed queries | reset

# Generate a more specific error event for output

Single | DontCont | SubStr | 3 subsequent failed queries |
Three subsequent queries have failed due to high CPU load |
shellcmd sendevent.sh "%s" | cpu_overload

# Generate a general error event for output

Single | DontCont | SubStr | 3 subsequent failed queries |
Three subsequent queries have failed |
shellcmd sendevent.sh "%s"
```

The following section describes an experiment for measuring *sec* performance.

6 Performance of *sec*

This section describes an experiment for measuring *sec* performance that was conducted in the Union Bank of Estonia. The experiment lasted for one week. Performance measurements were obtained from the computer that was running *sec* to monitor the banking card service.

The purpose of the card service is to offer the clients an opportunity to use banking cards for payment and for cash withdrawal. Card service consists of the following components - automatic teller machines (ATMs), point of sale terminals (POS terminals), and a card server. An ATM is a device that is primarily used for cash withdrawal by the clients. A POS terminal is a device that merchants rent from the bank, so that their customers can use banking cards for payment. The card server is a service process that receives requests from ATMs and POS terminals and processes them if the client who issued the request is authorized for it.

The card server keeps a detailed log about card service events (client requests, card server answers, card server internal errors, etc.), that served as an input for *sec*. During the experiment, new events arrived at a high rate - during daytime non-peak hours, about 3000-4000 new events (ca 300-400 KB of new information) were appended to the log every minute. At peak hours (between 4.00 and 6.00 PM), the arrival rate was 5000-6000 events (ca 500-600 KB) per minute.

Due to the high arrival rate of events, it was decided to put *sec* between the event source and the agent (and not between the agent and the network management server), in order to correlate events as early as possible. For security reasons, it was also decided that *sec* must not run on the card server machine, but on a separate computer and receive the card server log through a named pipe (card server log was sent over the network and written to the pipe by a separate shell script; see Figure 3).

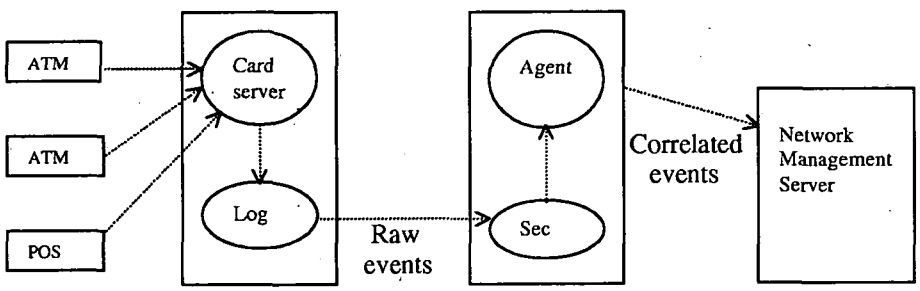


Figure 3: The experiment for measuring *sec* performance.

A low-end desktop computer with 200MHz Intel Pentium processor[5], 32Mb of memory, and Linux as an operating system was used for running *sec*. Since HP OpenView is used as the network management platform in the Union Bank of Estonia, the HP OpenView ITO agent was also installed on that machine. *Sec* was configured to use *opcmsg* utility for producing output events (*opcmsg* is an HP OpenView tool for generating events that are received by the local ITO agent). There were 63 rules specified in the configuration file of *sec*: 29 *Single* rules, 9 *Pair* rules, 1 *PairWithWindow* rule, 5 *SingleWithThreshold* rules, 17 *SingleWith2Threshold* rules, and 2 *Suppress* rules.

Figures 4-7 display data about CPU utilization and 1-minute load average of the computer that was running *sec* during the experiment[6].

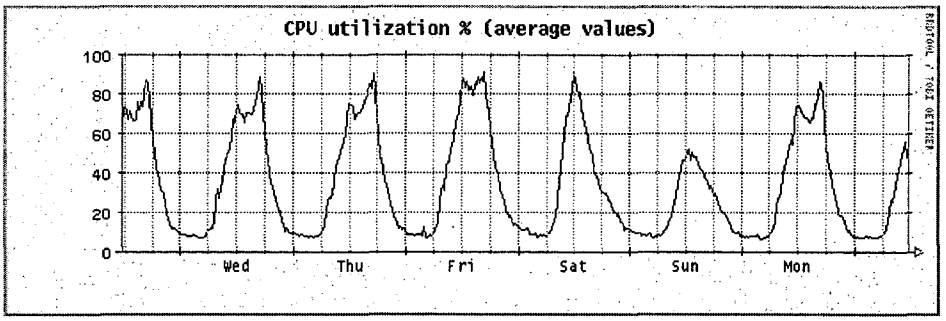


Figure 4: CPU utilization (average values).

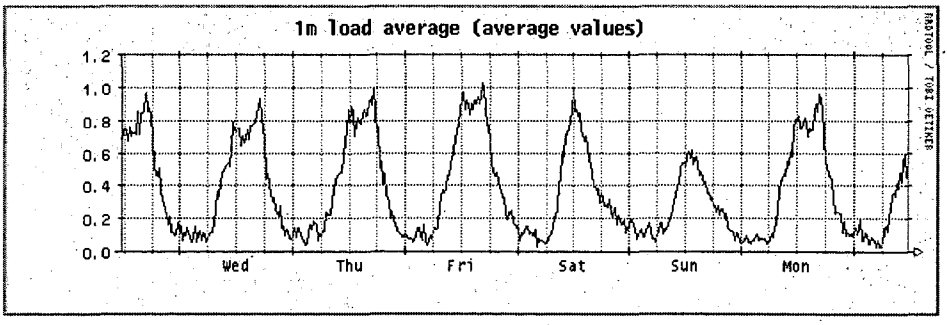


Figure 5: 1-minute load average (average values).

Graph data were gathered at 1 minute intervals. Since it was impossible to accommodate all data points to the graphs (due to their limited size), every line

[5]The processor did not have MMX support.

[6]Graphs were produced with RRDtool by Tobias Oetiker.

pixel represents the average value for 17 minute period in Figure 4 and 5, and the maximum value for 17 minute period in Figure 6 and 7.

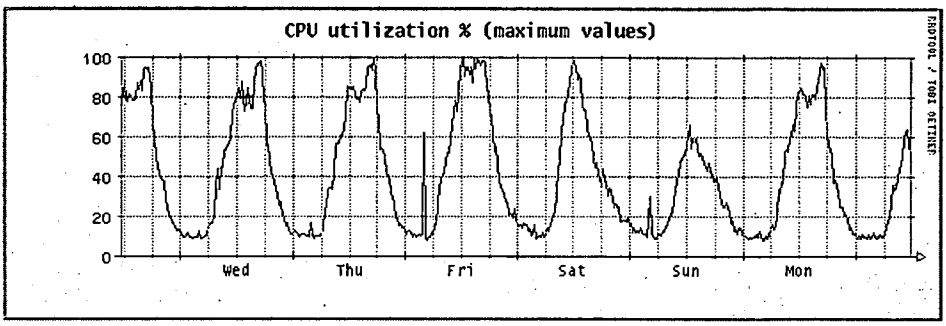


Figure 6: CPU utilization (maximum values).

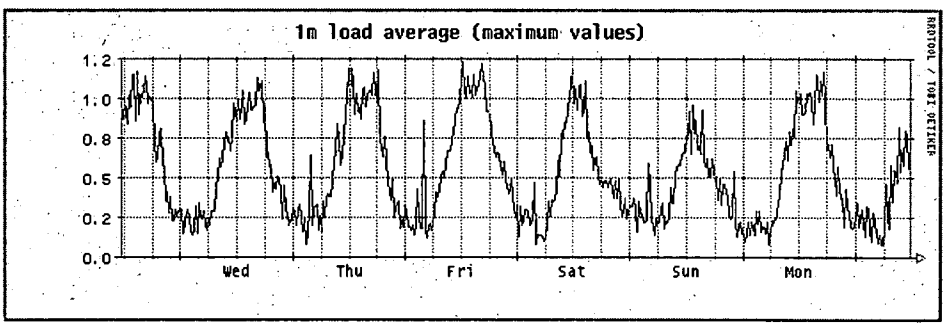


Figure 7: 1-minute load average (maximum values).

About 2.5 million events were matched by the rules during the experiment (that makes about 4 matched events per second as an average). Those input events were reduced to 101 output events by *sec*.

Performance data that were gathered shows that *sec* performs well under heavy event load. Though low-end hardware was used for conducting the experiment, *sec* was able to process input events in a timely manner without imposing heavy load on the local hardware resources. During the experiment, *sec* consumed only 4MB of physical memory. Despite the high arrival rate of events, 1-minute load average exceeded the level of 1.0 only at peak hours. Although the computer CPU was relatively slow, it was completely utilized only occasionally - in Figure 6, there are only 17 data points with the value of 100 per cent.

7 Current status of *sec* and availability

One year of experience with *sec* has proved that it is an efficient tool for correlating massive event streams locally. *Sec* has been successfully applied for network management in a number of companies in Europe and U.S., mainly in telecom and financial institutions.

It is difficult to estimate the total number of companies and organizations which are using *sec*, since no registration is required to download it. However, at the time of writing this, the *sec* download web-page had been visited more than 3,000 times. The author has received reports of successful use of *sec* on Solaris, HP-UX, Linux, and Windows2000 platforms.

In October 2001, the first version of *sec-2.0* was released, that implements a number of new action types, augments the properties of a context, and supports enchanced configuration file syntax.

Sec is distributed under the terms of GNU General Public License, and can be downloaded from *http://kodu.neti.ee/~risto/sec/*.

8 Conclusion

Although commercial network management platforms provide means for event correlation, there are still some problems that hinder the use of commercial event correlation engines - commercial engines are quite expensive, many of them do not work independently or with other network management platforms, and they also have a limited support for different operating system platforms. Since a lot of research has been done in the field of event correlation over the past few years, some non-commercial correlation engine prototypes have been created. There is, however, no freeware correlation engine available yet which would be mature enough for using in a production environment.

In this paper the author presented a free platform independent tool for local event correlation called *sec* (Simple Event Correlator), that implements a set of correlation operations essential in practice. *Sec* can be used as a correlation engine for the whole agent event stream, but also for individual file-like event sources.

For a future work, the author plans to use *sec* in other research areas which are similar to network management and which could benefit from event correlation techniques (e.g., intrusion detection).

Acknowledgments

Author wishes to thank the Union Bank of Estonia for supporting this work. Author also thanks Prof. Ahto Kalja for providing remarks and suggestions that helped to improve the quality of this paper.

References

[1] Cisco Systems. *Cisco Network Monitoring and Event Correlation Guidelines.* Reference Guide, Cisco Systems Inc., 1999.

[2] Catherine Cook, Budi Darmawan, Mike Foster, Stephane Gillardo, Vasfi Gucer, David Kong, Dinesh Kumar, Edson Manoel, Fred Plassman, Roger Reynolds, Kenshoh Sugitani, Samson Yiu. *An Introduction to Tivoli Enterprise.* Tivoli Redbook SG24-5494-00, IBM Corp., 1999.

[3] Paul Fearn, Raj Chityal, Nancy Jarin, Elise Kushner, Gordon Lilly, Darren Pike. *TEC Implementation Examples.* Tivoli Redbook SG24-5217-00, IBM Corp., 1998.

[4] Paul Fearn, Arne Olsson, Larry Bajuk, David Edwards, Peter Glasmacher, Gareth Holl, Istvan Szarka. *Integrated Network Management Solutions Using NetView Version 5.1.* Tivoli Redbook SG24-5285-00, IBM Corp., 1999.

[5] Boris Gruschke. *Integrated Event Management: Event Correlation using Dependency Graphs.* Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, 1998.

[6] Hewlett-Packard. *Event Correlation Services - Designer's Guide.* HP document J1095-90304, Hewlett-Packard Company, 1998.

[7] Hewlett-Packard. *HP OpenView IT/Operations Concepts Guide.* HP document B6941-90002, Hewlett-Packard Company, 1999.

[8] Hewlett-Packard. *Managing Your Network with HP OpenView Network Node Manager.* HP document J1240-90021, Hewlett-Packard Company, 1999.

[9] Stephen E. Hansen and E. Todd Atkins. *Automated System Monitoring and Notification With Swatch.* Proceedings of USENIX 7th System Administration Conference, 1993.

[10] G. Jakobson and M. Weissman. *Real-time telecommunication network management: Extending event correlation with temporal constraints.* Integrated Network Management IV, 1995.

[11] Mika Klemettinen. *A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases.* PhD Thesis, University of Helsinki, Finland, 1999.

[12] Wolfgang Ley and Uwe Ellerman. *logsurfer(1) and logsurfer.conf(4) manual pages.* See http://www.cert.dfn.de/eng/logsurf/

[13] G. Liu, A. K. Mok, E. J. Yang. *Composite Events for Network Event Correlation.* Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management, 1999.

[14] Dilmar Malheiros Meira. *A Model For Alarm Correlation in Telecommunication Networks.* PhD Thesis, Federal University of Minas Gerais, Brazil, 1997.

[15] Elaine Rich and Kevin Knight. *Artificial Intelligence.* 2nd ed., McGraw-Hill, New York, 1991.

[16] Stefan Uelpenich, Robi Banerjee, Peter Holm, Alain Queffelec. *Creating Custom Monitors for Tivoli Distributed Monitoring.* Tivoli Redbook SG24-5211-00, IBM Corp., 1998.

[17] Hermann Wietgrefe, Klaus-Dieter Tuchs, Klaus Jobmann, Guido Carls, Peter Froehlich, Wolfgang Nejdl, Sebastian Steinfeld. *Using Neural Networks for Alarm Correlation in Cellular Phone Networks.* International Workshop on Applications of Neural Networks in Telecommunications, 1997.

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János